In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

University of Alberta

The ANIMUS Project: A Framework for the Creation of Synthetic Characters

by

Daniel Torres

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2003

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisisitons et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-87898-8
Our file Notre référence
ISBN: 0-612-87898-8

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

Canada

University of Alberta

Library Release Form

Name of Author: Daniel Torres

Title of Thesis: The ANIMUS Project: A Framework for the Creation of Synthetic Characters

Degree: Master of Science

Year this Degree Granted: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Date: September 29/2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **The ANIMUS Project: A Framework for the Creation of Synthetic Characters** submitted by Daniel Torres in partial fulfillment of the requirements for the degree of **Master of Science**.

Date: September 26, 03

Abstract

We integrate research and concepts from biology, cognitive science, brain theory, artificial intelligence, and artistic design and animation, to create a framework called ANIMUS for the implementation of autonomous synthetic characters that convey the illusion of being *alive*. The framework includes a perception layer that allows the character to be aware of its environment and the user, trigger reflexes in response to stimuli, possess selective attention, and perform pavlovian conditioned learning. A cognition layer that provides tools for implementing high–level tasks like simulation of emotions, goals, social behavior, beliefs, knowledge, needs, and desires, and react in believable ways according to the character's personality. Finally, an expression layer that implements a flexible 3D animation engine similar to techniques used in traditional animation, allowing the character to express its inner states according to its unique personality. ANIMUS creatures closely follow an artistic conceptual design, favoring the creation of interesting and meaningful characters.

Dedication

To my parents and my family
To Sandra, and all of my friends

Acknowledgements

I would like to thank my supervisor, Dr. Pierre Boulanger, for his incredible support and guidance during these two years. I also want to thank him for all the resources he generously provided so that I could do my research using the best toys. In particular, I want to thank him for his economic support both during my stay in Canada and my trips to Europe and Japan (and all the nice visits to the Banff Centre).

I want to thank Conacyt for their strong and effective economic help. It is because of them that I am here in the first place. I have a firm commitment to my country and I hope to be able to give more of what I have taken in the form of education and technologic development.

I also want to thank Martin Jagersand for his most excellent friendship, advice, continuous support during my degree, his swedish fast food, and those many times he saved me from starving at night at the computing science building. My thanks also go to Walter Bischof for all the really big books and knowledge he shared with me, his fine editing and revision abilities, and his orientation on the fields of biology and neurology. I am also in debt with Lynn Schlecker for her professional editing advice and guidance through the mysterious ways of written english, and for her fine friendship. Many thanks to Pablo Figueroa for his help in my time of need when I first arrived in Edmonton, and for all the shared cups of coffee. It is also important to acknowledge the patience and help provided by Sandra, my girlfriend, during the time when I wrote this thesis; her support and continuously provided meals and attentions were fundamental for the making of this work.

Table of Contents

1 Introduction 1

1.1 Two Linked Concepts 2

1.2 Motivation and Importance 2

1.3 Not Only Art; Not Just Technology 3

1.4 The ANIMUS Project 4

1.5 Challenges and Contributions 6

1.6 How to Read This Thesis 7

1.7 A Definition for the Word "Emotions" 8

I Preliminary Information 9

2 A Survey of Previous Projects and Methodologies 10

2.1 A Collection of Techniques for Perception, Cognition and Expression 11

2.2 A Survey of Related Projects 16

2.3 Discussion about Related Projects 21

3 Basic Concepts and Definitions 25

3.1 Definitions 25

3.2 Setting Up a Virtual World 27

3.3 The Flow of Information in the ANIMUS Framework 29

3.4 Summary 30

II The ANIMUS Framework 32

4 Architecture 33

4.1 A Three–Layer Architecture 35

4.2 Some Claims about ANIMUS 37

4.3 Playing with Alebrije: A Practical Example 38

4.3.1 A Brief Conceptual Design About Alebrije 38

5 Perception 40

5.1 An Early Perception and Attention System 41

5.2 Perceiving Intensity 43

5.3 Perceiving Variation 45

5.4 Automatic Reflexes 46

5.5 Conditioned Learning 47

5.6 Attention Shifting 48

5.7 Creating a Perception Layer for Alebrije 49

5.8 Design Tools 53

5.9 A Simple Perception Simulation 53

5.10 Discussion 55

6 Cognition 58

6.1 The Role of High–Level Cognition Systems 59

6.2 Driver System 60

6.3 Dictionary of Entities and Characters 62

6.4 A Temporal Memory Model 63

6.5 Creating a Cognition Layer for Alebrije 64

6.6 Summary and Discussion 66

7 Expression 67

7.1 Common 3D Animation Systems 68

7.2 The ANIMUS Animation Engine 70

7.2.1 The Role of the Artist 75

7.2.2 The Role of the Designer and the Programmer 75
7.3 Creating an Expression Layer for Alebrije 76
7.4 Summary and Discussion 79

8 A Minimalistic Implementation **80**
8.1 Conclusions 85

9 General Comments and Future Work **86**

Bibliography **88**

III Appendices **93**

A Diagram of ANIMUS Classes **94**
A.1 ANIMUS Perception and Cognition Classes 95
A.2 ANIMUS Expression Classes 95
A.3 Abstraction Levels 96
A.4 XML Parsing objects and its derivates 97

B DTD's for the ANIMUS Framework Files **98**
B.1 Networks 98
B.1.1 Network DTD 98
B.1.2 Parameters 99
B.1.3 Example 99
B.2 Physical Layer Definition Files 99
B.2.1 Physical Layer DTD 100
B.2.2 Parameters 100
B.2.3 Example 101

C Framework Objects and Functions **103**
C.1 CAnimusCharacter Class Reference 103
C.1.1 Detailed Description 103
C.1.2 Constructor & Destructor Documentation 104
C.1.3 Member Function Documentation 104

C.1.4 Field Documentation 106
C.2 CAnimationThreadManager Class Reference 108
C.2.1 Detailed Description 108
C.2.2 Member Function Documentation 110
C.3 CArticulatedCharacter Class Reference 114
C.3.1 Detailed Description 114
C.3.2 Member Function Documentation 114
C.3.3 Field Documentation 115
C.4 CCharacterPoseEngine Class Reference 116
C.4.1 Detailed Description 116
C.4.2 Member Function Documentation 116
C.5 CDataRecord Class Reference 121
C.5.1 Detailed Description 121
C.5.2 Constructor & Destructor Documentation 121
C.5.3 Member Function Documentation 121
C.6 CDriver Class Reference 123
C.6.1 Detailed Description 123
C.6.2 Constructor & Destructor Documentation 125
C.6.3 Member Function Documentation 125
C.7 CEntity Class Reference 127
C.7.1 Detailed Description 127
C.7.2 Member Function Documentation 128
C.8 CMemoryUnit Class Reference 129
C.8.1 Detailed Description 129
C.8.2 Member Function Documentation 130
C.9 CStimuli Class Reference 133
C.9.1 Detailed Description 133
C.9.2 Member Function Documentation 133
C.10 CVirtualWorld Class Reference 136
C.10.1 Detailed Description 136
C.10.2 Member Function Documentation 136
C.10.3 Field Documentation 137

List of Tables

2.1 General perception techniques. 11
2.2 General cognition techniques by application. 13
2.3 Related projects and their general features. 24

3.1 Events triggered by the world manager. 31

5.1 General ANIMUS constants and variables. 42
5.2 Attention network constants and variables. 43
5.3 Variation module constants and variables. 43
5.4 Reflex agents and ART network constants and variables. . . . 44

6.1 Callback messages for ANIMUS drivers 61

List of Figures

3.1 Abstract layout of the immersive virtual environment. 28
3.2 Several configurations for a virtual reality environment. 28
3.3 Perception–related events triggered by the world manager. . . 30

4.1 Karmatron y los Transformables 34
4.2 The three main components of an ANIMUS character. 36

5.1 Architecture of the Perception Layer 44
5.2 Structure of an Attention Neuron 45
5.3 An ANIMUS character with a simple reflex agent that moves the body away from bullets 47
5.4 ART structure 47
5.5 ANIMUS Cross–layer attention shifting model. 49
5.6 Attention network graphic design tool 54
5.7 Graphic stimuli editor 55
5.8 The unmodified noise stimuli created by the insects. 55
5.9 Reaction to both stimuli by Alebrije's Attention Network . . . 56
5.10 Reaction to both stimli by Alebrije's Variation Module 56
5.11 The Alebrije character while sleeping (left), looking at computer's insect (center) and turning to face the user's insect(right) 57

6.1 Driver network graphic simulation tool 62
6.2 Two characters perceiving the same entity. 62
6.3 The ANIMUS Temporal Memory model. 64
6.4 Data referencing between temporal memory and dictionary instances. 65

7.1 Blended animation from Pose $w = y$ at Node x, to a virtual pose $w = 0.7$ at Node n. 71
7.2 Obtaining a bezier interpolation with two linear interpolations. 72
7.3 Time and acceleration in easing–in and easing–out animations. 74
7.4 Alebrije, from several different perspectives. 76
7.5 Alebrije's bone layout. 77

Chapter 1

Introduction

"I am enough of an artist to draw freely upon my imagination. Imagination is more important than knowledge. Knowledge is limited. Imagination circles the world"

-Albert Einstein

It is likely that more than once, while reading a good novel or story, we stop for a moment to wonder how powerful the imagination of the writer must be in order to create such an extraordinary repertoire of characters. Most of us probably remember at least one imaginary fellow that, in spite of knowing it to be unreal, claimed life from the pencil of its author and caused a profound impression in our minds –sometimes our lives–, to the extend that we willingly imagine it not as the product of the author's imagination, but as something that *lives* on its own. Indeed, the ability of the author to make his creation walk, speak, act, feel, love, fear, conspire, and dance trough the pages of a book is so finely done that the character stands for itself, is perfectly believable regardless of how fantastic or supernatural it might be. Has any animated film character gained a place in our heart? Has any imaginary creature (human or otherwise) ever taken us to experiment sadness, joy, anger, or any other feeling we have felt for *real* human beings? How is this possible? Thomas and Johnston[68] may enlighten us:

> "Conveying a certain feeling is the essence of communication in any art form. The response of the viewer is an emotional one, because art speaks to the heart. This gives animation an almost magical ability to reach inside any audience and communicate with all peoples everywhere, regardless of language barriers...(p. 15)"

Imagine that you were granted a ten–minutes audience with your favorite character (be it from a book, an animated film or any other art form). Both of you are left alone in a room. This time it is not your imagination: the character seems to *know* you are there, tries to *communicate* and *interact*

with you, speaks to you, reacts to your actions and behaves just as you thought it should. This dream of bringing imaginary characters to "*life*" has inspired the work of many people belonging both to the art and scientific communities.

The framework presented here provides a base for the implementation of characters that look, react and behave as if they were alive. This thesis integrates knowledge in computer graphics, Artificial Intelligence, theory of the mind, and cognitive science under the vision and creative force of Art, in order to create characters that not only are robust and functional, but also interesting, believable and able to develop a fruitful immersive interaction with the audience. In other words, the objective of this thesis is to *support and contribute to the creation of interactive characters that seem to be alive.*

1.1 Two Linked Concepts

There are two concepts that need to be defined due to the frequency in which they are used in this thesis, and their importance for understanding our goals. We borrow from Reilly[57] and Bates[31, 32] the attribute of *believable* applied to an artificial agent that, like a human actor in a theatrical play, convinces the audience of its authenticity. Every gesture, every action and movement, every denoted feeling, emotion, and intention, are congruent with the actor's role in the play. We are willing to *believe* the actor is in fact the character it represents. This depends, of course, on the ability of the actor and subsecuentially of the virtual agent in the immersive environment. The disposition of the audience to ignore the fact that the actor is actually an actor, and see him/her instead like the character he/she represents is called *suspension of disbelief.* It is important to note that, as pointed out by Loyall[39] and mentioned in the next section, believable does not necessarily mean *realistic.* Often the challenge resides precisely in the task of making an *imaginary* or *fantastic* creature believable to an audience. It is here where art comes into play. For an excellent discussion of this and other related topics please refer to Thomas and Johnston's masterpiece: *"The Illusion of Life"*[68].

1.2 Motivation and Importance

Synthetic characters have a wide range of applications that go from virtual drama[44, 31, 57], videogames, artistic interactive installations[18], and other art–related projects of diverse kinds, to therapeutic tools, help and support systems[2], human training tools[42, 23, 24], artificial life simulations[66, 13], computer vision, and practically any project where a more natural interface between the human and the computer is desired. Characters that look and feel more natural, and have the ability to simulate emotional behavior and personality, have less difficulty in obtaining the attention of the user, making the interaction more helpful, effective, beneficial and interesting. In addition, synthetic characters offer a good testbed domain for experiments related to psychology, social sciences and human personality and behavior.

An effective synthetic character is helpful both for artists and scientists. In

a world where computers constitute an everyday tool, lifelike agents that simulate emotions and personality promise a more natural interface for the next generation of advanced man–machine applications. This work explores methodologies that will contribute to projects that are artistic in nature, particularly those related to immersive virtual environments, artistic installations and videogames. Nevertheless, the proposed framework provides a robust start for creating synthetic characters applicable to other domains as well.

1.3 Not Only Art; Not Just Technology

> "Fantasy, abandoned by reason, produces impossible monsters;
> united with it, she is the mother of the arts and the origin of all marvels".
> – *Francisco Goya*

Artists know how to create rich characters for media like books, drama, and film, but they hardly possess the ability to create a robust interactive creature that maintains the same attributes (memory, personality, emotional and rational behavior, intention, beliefs and goals, plans and actions) in a real time virtual environment.

Scientists, on the other hand, have the knowledge and technology to make interactive agents, to integrate advanced AI and automation techniques that simulate biological or intelligent behavior. In most cases they obtain characters that look more *intelligent* than *believable*: too perfect, predictable, or robot–like to be interesting at all for the audience. These agents can play chess or "efficiently" learn to drive, they can find the only way out of a maze or search through millions of options to discover the best strategy to solve a problem, they can mimic the exact movement of a real bird. Still, a believable character ***is not necessarily intelligent***; it can be tricked; it is not always efficient, nor optimal or even realistic. As Reilly[57] points out: *"It turns out that sometimes being more realistic can decrease believability. For example, watching extremely realistic animation of human faces, like that of Terzopoulus[66], can be very disturbing, whereas watching unrealistic animation, like Charlie Brown, can be very satisfying"...(p. 11).*

It is clear that the works of science and art must be combined in order to create a successful believable character. The writer knows how his character thinks, feels, learns, moves, reacts and behaves, while the scientist knows how to put that knowledge into an interactive virtual agent. Without each other, the writer will only create books, and the scientist will only make robots.

Numerous researchers have already begun to explore this powerful synergy between arts and sciences. In Chapter 3, we will review what has being achieved so far.

1.4 The ANIMUS Project

Animus is the latin word for *Mind*. According to the archives of the University of Notre Dame, it also refers to the character, intellect, memory, courage, vivacity, and will. Jung places this concept as the perfect counterpart of *Anima*, just as *Ying* is the complement or the opposite for *Yang*. Animus is a masculine concept that denotes the presence of consciousness and rationality; *logos*, as Jung applies it to the psychological nature of men. Anima (feminine in nature) relates to the soul, and the feelings or emotions as autonomous and chaotic beings. Since both concepts seem to drive the behavior of a human being it is only natural to choose one of them as the name for a framework that intends to create the illusion of life on artificial characters. Selecting Animus instead of Anima is an obvious choice, since this framework tries to mimic *physical*, *biological*, and *rational* features, that is, tangible elements that conform the necessary foundation for the more intangible ones, or Anima. The framework provides a *body* in which the designer is able to put a *mind* and ultimately a model of personality and emotions.

Implementing a model of emotions and social behavior, investing our synthetic character with a certain *personality*, are indeed elements that could very well fall into the area of *Anima* because they are not as tangible as the logical set of elements that drive an essentially rational behavior. Their creation is not direct responsibility of ANIMUS, but the framework is designed to provide the necessary elements for their existence. This is analogous to the body and the mind, which are themselves a reservoir for more complex elements (personality, emotions, role, intention) to manifest by means of a behavior that seems to go far beyond the purely biological existence of an organism, bringing the ability to imagine, to improvise, to create and invent, to generate chaos.

ANIMUS is a framework that constructs the body of a synthetic character. Having such body, it is possible to implement ANIMA. *The ultimate goal of my research is to have architectures for both elements*, ANIMUS was designed with ANIMA in mind from the very beginning, and one of the main goals of every component is precisely to promote or convey an emotional state to the audience. Nevertheless, specific models for personality and emotions will not be discussed in this thesis, first because it is up to the designer to use the tools provided by ANIMUS and his/her own research to implement the model that best applies to each specific character; and second, because such models by themselves would fill the pages of another thesis.

It is necessary to stress the fact that although ANIMUS makes it possible to create characters that simulate emotions and personality, such models need to be implemented by the user because of the following reasons:

- There is no simple solution to the problem of implementing emotions and personality for a synthetic character. Several factors like the domain, the character's nature and the particular characteristics and objectives of every project will affect the requirements for such models.
- Different characters will have different needs in terms of their emotion and personality systems. Instead of forcing the user to apply a particular model, We provide the necessary tools for him/her to implement the model that best solve such needs for each particular case.

The main goals of this project are:

1. Make it possible to create believable virtual characters that interact with the user in an immersive 3D environment.
2. Provide means for art to define, influence, and directly contribute to the creation process, so that the final result complies with the character in the mind of the artist.
3. Provide resources that facilitate the implementation of a model of emotions and personality, and an integration level that makes it easy to design a character that conveys such emotions to the audience. The designer must be able to focus on the creation of such models without having to worry about the performance of the *body* of the character. At the same time, the framework must provide all the necessary information for such models to work.

To simplify this challenge, we picture the virtual character as a *living* creature and ask ourselves:

- How is this creature perceiving the world? (What and how is it perceiving the environment?)
- What happens inside its mind?
- How does it express its inner thoughts, emotions and intentions graphically to the user?

The first question is the problem of *perception*, and implies the way in which the character is aware of the world. Without a good perception layer there is hardly any interactivity at all, and preserving the believability of the character depends on the authenticity of its perceptive ability. For example, a character that is supposed to be blind should not react to light or visual events, while a character with a fine hearing sense must proceed in a different way to those with "normal" hearing abilities. Similarly, two characters separated by a wall should not see each other. This principle is called *visual honesty* and will be discussed in Chapter 3.1.

The second question relates to *cognition*. It is here where we apply different models of emotions, goals and personality. It also includes planning and decision making, social and individual behavior, knowledge representation, and different forms of rational learning.

The third question refers to the problem of *expression*. This is basically the graphic incarnation of the character in the virtual environment. The character's animation must be adequate to suspend the disbelief in the audience and convince them about the authenticity of its emotions, intentions, inner states and actions, reflecting its own personality. In other words, the movement of a character is analogous to the movement of an actor: both must reflect their role. No two characters move alike, and the same character might perform the same action in different ways according to the situation. Probably no other dimension of the character will affect its believability as the expression does.

ANIMUS proposes a framework composed by three layers: perception, cognition and expression. One of the main differences between ANIMUS and other frameworks is this tree–layer modular approach. As we will see later, many architectures perform perception and cognition tasks at the same level, or their expression is bounded only to their cognition systems. The implications and consequences of that approach will be analyzed in depth later on.

1.5 Challenges and Contributions

Complex believable characters (those that gave the illusion of having their own personality and emotional behavior) started to exist in early animated films, even before the addition of sound to movies. At that point, the challenge of interactivity was not present due to technological and scientific constraints. As time passed by, the possibility of investing synthetic characters with lifelike features became a source of inspiration for some scientific and artistic communities. Although today we can find excellent projects devoted to the creation of synthetic characters, most of the projects strongly oriented to art fail to provide an optimal degree of interactivity, while the ones oriented to science lack the spark of life that only artists can provide.

Several related projects are reviewed in Chapter 2, they are some of the best examples to be found at this time regarding synthetic characters. Still, most of their architectures are strictly bounded to a particular domain or application (some only work with help systems, others perform in very specific scenarios like a pub or a cafe, and there are ones that exist for the sole purpose of simulating the way an animal moves or hides away from predators); their functionality is deeply limited to a certain platform or technology (they only work with the graphic library Y, a graphic environment X, or a given hardware platform; some produce facial animation but cannot control a body; their emotional behavior is acceptable but the interface with the user is text–based); their modules are so highly coupled that it is difficult to modify the functionality of a part without affecting the system as a whole or completely destroying the advantages of the project itself (they can only implement certain model of emotions, or their behavior is purely based on rules and cannot be changed; the whole architecture depends on how the character perceives objects, or how it animates), therefore it is necessary to completely understand their inner functionality, which leaves little or no space for adequating an artistic design to the functionality of the character. Most of them rely on technology chosen at that time and cannot be easily adapted to new or novel input and output devices (some produce text output, and porting them to a virtual immersive environment by itself would create a completely new project; others will accept input from certain devices only, in a format that would make it difficult to adapt an artistic installation where the information comes, let's say, from a galvanic skin resistance sensor); their inner AI is based on a strict algorithm or methodology that cannot be changed without transforming the architecture into a completely different system. Some projects present an option that might be optimal for a certain character, but too complex or insufficient for another, in which case the only solution would be to work with different architectures or sacrifice one for the other.
The proposed framework presents a series of advantages over most of those

problems. It is highly modular, so many components can be replaced as needed by the user. This also makes it easier to port the framework to other domains and apply different technologies without affecting its parts. It provides a flexible input and output interface, so designing a character that reads input from the keyboard becomes as straightforward as creating a character that reads the user's pulse, detects the sound produced by another character, or tracks the user's movement by means of magnetic sensors. It is not bounded to a particular AI algorithm, but provides tools for the user to implement any model of emotions, decision–making, or other high–level architecture, which facilitates the evolution of the character's internal cognitive systems without compromising the rest of the architecture. The architecture favors the straightforward creation of characters with simple behavior (like animal–like creatures powered by rule–based systems, for example) but provides a solid base for doing research with more complex systems that implement models of memory, emotions, personality *et cetera*.

One of the most important contributions of this framework is the ease in which a conceptual artistic design can be transformed into a fully functional synthetic character. Artistic definitions and constraints are present at every module of the system, shaping the way the character perceives the environment, reacts, and interacts with the user and the virtual world. In fact, the production cycle for a synthetic character using the ANIMUS architecture assumes the previous existence of an artistic design, because most of the character's configuration parameters will be derived from it. Another important contribution is the framework's flexibility and simplicity: there is no need to create complex AI elements for simple characters that only need a rule–based system (like animals or objects), although the designer is free to perform his own research and implement more complex cognitive systems, or apply technologies found in other research papers.

Other contributions include a pose–based animation engine that provides a flexible interface to the character's body, making it possible to create non linear animation sequences or generate new animations in real time, following the character's artistic constraints and definitions. We will also describe a selective attention and perception system based on functionality observed on biological systems, which promotes a more natural and believable perceptual and reactive behavior. Tools developed for the high–level cognitive systems include a temporal memory model that facilitates the implementation of novel cognitive features like experience–based learning, interpolation and extrapolation of world phenomena, knowledge modeling and storage, among others.

1.6 How to Read This Thesis

Chapter 2 contains a survey on related research. This chapter has two main objectives: First, it provides a background on the state–of–the–art relating synthetic characters, and second, it offers a source of information for those interested on creating their own architectures or modifying the ANIMUS framework to fit their needs.

The ANIMUS framework itself is described in Chapters 4 to 7. Chapter 4 introduces to an example project named Alebrije that will be used to

exemplify the application of the framework. Chapters 5, 6, and 7, begin by describing the principles and methodologies that constitute the foundation of each layer, and finish with a more technical explanation on how to use the framework to implement each one of Alebrije's modules.

Chapter 8 contains a step–by–step tutorial on how to create the basic C++ class skeleton for an ANIMUS character. It describes the minimal declarations and structures necessary to implement basic support and integration between layers. The intention is to provide a helpful example to the reader who wishes to use the framework for his/her own projects.

The appendices constitute a reference manual for the low–level and programming details of the framework, including DTD definitions, listing of functions, diagrams of classes and objects, and in general all kind of technical information related to the core implementation of the ANIMUS layers.

1.7 A Definition for the Word "Emotions"

Throughout this thesis we will constantly refer to the *emotions* of a virtual creature or character inside the computer. This is not the same concept of emotions as applied to a human being. Emotions in humans is a vast topic that embraces many disciplines ranging from the merely psychological to the purely spiritual. It is not our intention to discuss the nature of emotions in human beings, and at the same time we are not claiming that our virtual characters can *experience* emotions as we do. When we speak of emotions in a virtual character this must be understood as the *illusion* of emotions or *pseudo–emotions*; the character is *imitating* a human emotion. When applied to humans, we will not refer to their generation process either, but to their sensible experience.

Part I

Preliminary Information

Chapter 2

A Survey of Previous Projects and Methodologies

"Dios mueve al jugador, y éste, la pieza
¿Qué Dios detrás de Dios la trama empieza
de polvo y tiempo y sueño y agonías?"

-J.L. Borges, Ajedrez

The projects detailed in this chapter have been a source of inspiration to our own work. It is hard to start a project as complex as the design of a synthetic character without previous knowledge about what has been done before. Reviewing related work is important to visualize our own work and discover our limitations and strengths. We begin this chapter by describing a series of known techniques for the implementation of components for synthetic characters. Understanding such techniques is an excellent introduction to the survey of related projects, which in turn creates an appropriate setting for presenting the general idea of the framework.

Our intention is to provide an useful reference for those who wish to apply existing techniques (including the ANIMUS framework) for the design of their characters, or even better, as a starting point for their own individual research. Since it is hard to explain all the details of each project, only essential information will be provided. The reader can refer to the bibliography for a more concise explanation. At the end of the chapter we discuss the advantages and disadvantages of the presented projects.

2.1 A Collection of Techniques for Perception, Cognition and Expression

General perception of the virtual world can be considered at three different levels: internal, external, and simulation of hidden states in other characters. This last option is related to the theory of mind as described by Gordon[22]; a high–level cognition process where we try to infer the hidden state of other characters to help our own decision process according to our goals and objectives. Internal stimuli are concerned with sensations or features generated *inside* the character. Examples are pain, hunger, sexual pleasure, anger, and tiredness. External stimuli are perceived from *other* characters[1]. Examples of external stimuli include the position of an entity, its velocity, color, and sound. Hidden states are features that cannot be directly perceived since they remain *inside* other characters, but can be inferred by correlating external stimuli with our own knowledge, and logical or rational structures. For example: the emotional state of other characters, internal physical state, etc. The perception of internal and external stimuli, plus the hidden state inference system of our character provide the necessary input for producing a response. Examples of perception systems can be found in Kline[34] and Isla[16].

General Perception Techniques			
Method	**Internal**	**External**	**Hidden**
Driver systems, driver networks	*		
Automated Segmentation		*	
Constant or *a priori*	*	*	*
Function based	*		*
Blackboard systems		*	
Query based		*	*
Sensor based	*	*	
High–level Simulation and modeling	*		*

Table 2.1: General perception techniques.

Table 2.1 presents a general summary of methods for all the three kinds of perception. This information is based on tendencies observed in related research; the reader must never assume that these are the only techniques, nor that their application is limited to the suggestions in the table. In fact, each project applies each method in a different way, and often a combination of methods is required to achieve the expected results. We will now briefly describe these techniques. Please refer to Table 2.3 for details on which projects implement the various techniques.

A *driver* is essentially an accumulator that keeps track of a certain magnitude or represents the state of a feature. Drivers are generally used as independent objects or as correlated elements inside a weighted network. They can be controlled by an external monitoring function or contain their own grow and

[1]In our virtual world architecture, it includes whatever information we can read from the blackboard or obtain by directly querying such characters.

decay behavior. A driver can represent the level of emotions like joy, anger, or frustration; sensations like hunger, pain, or tiredness, and virtually anything that can be measured. Two excellent examples of systems that make extensive use of drivers are *The Sims*[6] and Velázquez's[71] *Cathexis* model. In The Sims, virtual characters have a set of needs that increase as a function of time or other factors, and a wide array of actions available for the user to perform when the magnitude of a driver is dangerously high (the user must feed a hungry character, or entertain a bored one). Cathexis will be later explained.

An *automated segmentation* system applies one or more specialized algorithms to identify different elements, objects, and properties, in the perceived information. When the character gets a *snapshot* of the world state, it is necessary to distinguish the different elements that conform a scene. In most projects, such elements (like the number of characters in a room, the position of a table, the name of the closest character) are directly fed into the creature, but authors like Isla and Blumberg[16] prefer to let their characters decide for themselves and learn to classify new or unknown entities. This approach provides a higher degree of realism in the character's behavior

Some projects like Rousseau's[62] are focused on exploring the interaction between characters that are predisposed in different ways, and are less concerned with real–time behavior adaptation. One might want to design, for example, a system that generates scripts for TV drama. The designer creates all characters and decides how they relate to each other *before* the simulation starts (who hates whom, and which characters are close friends, for example). The simulation is then run and the consequences of the *a priori* relationship are evaluated. Some variation can be introduced and the simulation is run again to see how it affects the overall result. Another example is a simulation where character A is considered a natural enemy of B. During the experiment, B will always try to run away from A without having to infer how dangerous A is. These systems provide a practical and straightforward approach, but run the risk of becoming deterministic and linear almost immediately.

Function–based perception allows the character to deduct or define several features about itself and other characters by using available stimuli as input to one or more functions. For example: perceiving the position of some creature at different times lets other characters deduct its velocity. Desperation, for instance, can be a function of hunger, pain, and tiredness, which are features that can be read from their respective drivers. Examples of these systems are described in Velázquez[71] and Isla[16]

High–level simulation and modeling implies a series of techniques that give our character the ability to reason about its mental states and those of other characters. These techniques are relatively new as some are inspired by human cognitive processes, which are still open research in neuroscience and theory of mind. Some tasks performed by these systems are analogy, common sense, explanation, and intuition. These processes are usually linked to high–level cognition systems. Examples can be found in Gordon[22], Gratch[23], and Marsella[42]. A more detailed analysis will be provided in Chapter 6.

Blackboard systems use an organized information source that maintains updated data about the world. Instead of letting the character label or distinguish the objects and attributes of elements in the environment by itself, the

blackboard *tells* the character whatever it needs to know (or can be known). This simplifies the cognitive process because it is not required to use complex AI or analysis techniques to interpret the perceived information. Isla[16] describes one of these systems for their C4 architecture.

Query systems are similar to blackboard systems. Required information can be directly given to the character without having to apply complex analysis techniques, but it is the responsibility of the character to *ask* for such information. A good example is Scott[57]. Automated segmentation systems might perform a complex analysis over the appearance of a given character to decide what kind of animal it is perceiving, while a query system will directly ask the character or the appropriate information source.

General cognition techniques by application	
Application	**Techniques**
Emotion triggering	Goal–based, driver–based, state machines, hand–coded
Personality definition	Emotion–dependent action gradients, cross arousal and cancellation, hand–coded
Behavior selection	Behavioral features, rule systems, goal–based, plan–based priority systems, conditional triggering, behavior agenda, weight–based, emotional state
World Representation	Memory models, state machines, interpolation and prediction (internal and external),

Table 2.2: General cognition techniques by application.

Table 2.2 illustrates several methods for cognitive activities. Remember that cognition refers to processes that determine the character's goals, decision making, emotions, personality, and individual or social behavior. In a *goal–based* approach, actions are determined by the character's goals or objectives, the nature of emotions depend on how they relate to current goals. For example, a policeman will trigger frustration or fear in a thief that is about to rob the bank. A *driver–based* approach like the one in Velázquez[71] and Kline[34], holds the magnitude of emotions in a driver system. The presence of a policeman perceived by our thief, for example, will cause his *fear* driver to increase, but we will only see him trembling or running away if the magnitude of such driver exceeds a certain threshold. Additionally, evaluating the likelihood of success in robbing the bank might use the current fear magnitude as one of several input factors; it may also be used to alter the thief's skill level.

State machines are more deterministic structures. They imply a clear definition of the character's emotional state at each time step, and provide a reference for selecting actions based on its current mood. If the character is hungry he will look for food, if he is depressed he will perform a *sad–walk* instead of a *happy–walk*. Some systems use *hand–coded* rules for triggering emotions, examples include the works of Funge[30] and Rousseau[62]. In this approach, there is a more or less complex rule system that determines the character's emotional behavior according to conditions like the place, number of characters, social situation, nature and identity of others, current goals or

emotional state, and such. These systems tend to become deterministic and quite predictable after a short while, but they are easy to implement and might be useful when combined with other techniques, or when applied to certain types of characters like animals or creatures with simple behavior.

Defining the personality is a crucial step in the creation of a believable character. Some authors like Rizzo[50] propose a predefined collection of actions that are chosen according to the character's mood. Since the visible behavior of the character has to be congruent with his style and personality, to design a number of off–line *predefined* actions is easier than to model them on–line. Reilly and Bates[57, 31, 32] describe a system that elicits emotions and allows to set different behavioral features for each character to preserve their individual personalities. In other words, two different characters might experience sadness, which is essentially the same emotion, but each will react or behave in a different way according to their own personality. Other authors like Kline[34] suggest cross arousal and cancellation techniques, where the magnitude of joy, for example, will diminish the amount of frustration or sadness. In this case, personality is defined by the relationship between emotions. Also, one can define characters with more or less predisposition to certain emotions in different situations and with respect to different characters. and thus create, for example, a clown that is seldom sad, or an aggressive character that is usually mad at every event or some particular characters. Rizzo[50] proposes a framework where a priority profile will favor some actions and inhibit others.

As with emotion triggering, the personality of a character can be constructed by a set of *rules*, with the same disadvantage: behavior becomes repetitive, static (will not change or evolve given the experience of the character in the virtual world) and eventually affects believability. Nevertheless, when we are dealing with characters with simple personalities, or our expectations are not very demanding (like designing an animal or a robot), this last option is relatively easy and straightforward.

Regardless of the technique, it is desired that emotions influence the behavior and decision making process of our characters. *Behavior selection* is a very important feature. A fearless hero must be able to think clearly in dangerous situations, whereas a mouse will present a more believable behavior if it cowardly runs away when the cat is present. Most of the time it is also important that emotions affect performance. A nervous character must behave erratically or commit more mistakes than when he feels quiet. It is reasonable to think that our characters will have to choose between more than one behavioral option at a given moment. Imagine, for example, that our thief is surprised by the policeman while robing the bank and has the option of either showing fear, or anger. Reilly[57] proposes three different methods for dealing with situations like this: A winner–takes–all approach will take our character to choose the strongest emotion and act accordingly (our thief may be so afraid he will just start to tremble and drop his gun); an additive approach where two related emotions are combined to create a stronger reaction (fear plus anger can result in desperation), and a logarithmic combination, which produces results similar to addition, avoiding multiple emotions to generate intense reactions too quickly. The overall behavior will be determined by the choice and combination of methods used for emotion and personality representation. Either dictated by the character's goals, priorities, a set of rules or conditions, or an *execution agenda*, one must carefully see that the result is

a believable behavior that clearly conveys a strong personality to the audience.

The last challenge, and probably the most important, is *expression*. Most reviewed projects have a text based interface and interact with the user by means of natural language or a series of commands[2]. In a virtual reality environment this is not possible, as we need our characters to graphically *express* their inner emotional state, intention and personality to the audience. Regardless of the animation technique, we face the problem of selecting *what* to animate. Imagine the mouse is peacefully eating a piece of cheese when the cat shows up; should it drop the cheese and run? should it run while holding the cheese? should it keep on eating while running? should it valiantly face the cat? The following list describes several techniques for selecting and executing actions and animations; one may need to combine or adapt more than one to obtain the desired behavior for each character.

- **Independent Features**
  A straightforward technique to code animations is to create one set for each specific action, with several variations to reflect the character mood and/or behavior. A *gaze* action, for example, might include *angry-gaze* and *friendly-gaze* versions. These elements are explored in Reilly[57] and Rousseau[62]. It is also possible to have parametrized sets of actions: instead of two types of walk one could call some $walk(x)$ function where x is an *emotional* magnitude. Whatever the case, the chosen action will have a corresponding expression element. Needless to say, $run(x)$ for character A and B should produce different animations to reflect their distinctive personalities.

- **Sequential Execution**
  The most simple approach consists on a sequential list or queue of actions that get executed one after the other. When the mouse sees the cat it drops the cheese, then opens its eyes in amazement, then starts to run away. Lozano[46] describes an interesting system using this technique, and Cavazza[41] elaborates on a very original application for animations of narrative interest.

- **Hand-coded**
  A common and relatively easy approach is to hand code the character's expression system by using a set of rules. A good example is detailed in Funge[30]. If the cat is not present, eat the cheese. If cat is present abandon cheese and run for cover. As mentioned before, rule systems are useful when our characters are simple and their personality is not demanding (animals, facial gestures for basketball players in a videogame), however, their highly linear and predictable behavior will fail to interest the audience almost immediately.

- **Sequence Graphs and Verb Graphs**
  A better option is to design a graph where each node is an action or animation, and each arc encodes animation parameters or reference information. For example, from state A (eating cheese) to state D (running for your life) there might be several possible animation sequences. If time is of the essence (the cat is present) we could follow the shortest path; considering that arc weights measure time delay. It also allows better

[2]See Table 2.3.

control between animation transition. If our cat is sitting, it would not look believable to see it immediately running for the mouse, therefore its sequence graph is designed such that from state A (sitting) to state C (chasing) we always have to go through state B (standing up). In other words, we can design transition animations between states, order the engine to reach a particular state (what to do) and let it decide the correct animation sequence (how to do it). This approach is widely used in videogames. Another possibility is to have several layers of graphs to allow concurrent animations. Again, conflicting animations are avoided and adequate sequences are favored by the graph interconnectivity, or coded in the weights of the arcs. An excellent example of verb graphs can be seen in Isla[16].

- **Cross Inhibition**
  As mentioned in the last paragraph, there are sequences of animations that should not be executed neither sequentially nor concurrently (speaking and eating, running and crouching, sitting then jumping...). Our animation engine should have a way of handling this problem, for example, by transition states to allow adequate intermediate animations. Examples include Isla[16], Reilly[57], and Nicholson[5].

- **Goal–Produced and Prioritized Actions**
  In goal–based or plan–based architectures like Gratch[23, 24] and Rizzo[50], the character has a clear *notion* of what must be done, therefore actions and animation sequences can be organized and prioritized. If the mouse realizes its life is in danger, the piece of cheese is left behind when our character runs to a safe place. If eating is, however, an important goal (the mouse will starve to death if it does not eat that piece of cheese), the mouse might show itself nervous and repetitively stare at the cat and the cheese in desperation, until choosing between one of them is strictly necessary. In an extreme situation, our poor mouse will try to save the cheese, or stay there (if the cheese is too big to move), closing its eyes and expecting the worst...

The designer must consider the nature and capabilities of his characters and choose an adequate combination of perception, cognition and expression techniques. Simplicity must be always a rule, it is unnecessary to create a complex AI system for a simple character whose behavior can be expressed by four or five rules, while it would be hard to use a rule system for a character with rich personality that interacts with the user and grows with its experience. As a general guideline, no character should be programmed without having done an adequate artistic conceptual design first. If we are not sure about the character's behavior in certain situations, or it seems hard to select an adequate set of techniques for the creature's inner mechanics, we must spend some more time thinking about character as a concept before even touching the computer.

2.2 A Survey of Related Projects

It is impossible to mention every relevant project about synthetic characters. New and interesting research is being conducted around the world. The survey

presented here describes the projects that we consider to be some of the most representative. Table 2.3, compares the characteristics of each project.

Among the early works with interactive characters we find the Oz project[31, 32], where the term *believable* was taken from the art community and applied to computer agents to create characters for interactive drama. Agents in this environment integrate a broad range of abilities like individual personality traits, social and language skills, intelligent inferencing, and goal–oriented behavior. For this reason their authors call them *broad agents*. Oz agents implement a cognitive model of emotions based on Ortony[3]. Personality is modeled by the character standards (what makes it happy, sad, proud...), and the architecture implements behavioral features to perform actions in different ways according to a current emotional state. Emotions are goal–based, and related to the various ways in which the environment and the behavioral features success or fail at satisfying the character's goals. It is, for example, defined that the possibility of failing will trigger frustration, and how such feeling is expressed depends on the behavioral features of that character in particular. Every character follows a sense–think–act cycle. At each cycle, the sensed information is kept on the integrated sense model, or ISM, which provides the current representation of the physical world. External events are compared with goals, actions with standards and objects with attitudes. All emotions are elicited as a response to those relations, for example, hope exists when a goal is likely to succeed, happiness when it actually succeeds. All goals and their plans are kept in the *Active Plan Tree*. This tree is subject to two operations: success tests and context conditions. Success tests verify if a goal has been accomplished, in which case is erased from the tree. Context conditions test the result of a plan. If it fails is removed from the corresponding goal and other alternate plans are executed. When the character chooses plans for accomplishing a goal, only those that once succeeded are considered. The Oz project makes an excellent implementation of goal–based emotions and behavior, as an alternative to common driver–based approaches. Its model of internal goals and plans seems effective and simple.

Reilly[57, 73] retakes the work of the Oz project; and explores emotion representation, social behavior, and negotiation for synthetic characters. Emotions are stored in a hierarchical tree (an emotion can be defined as a child of another emotion type), allowing special queries like the combined magnitude of a certain emotional branch (how happy am I?), the combined magnitude of emotions directed to some target (how happy am I with X?), relations between magnitudes and goals (how happy would I be if I fail the test?), list of targets given emotions (who am I angry with?), list of emotions in order of magnitude, among others. Each emotion has a unique function that calculates its magnitude and decay, an emotion generator that specifies its triggers, and a place in the emotional taxonomy tree. In addition, characters try to infer the internal state of other characters and predict their plans and goals. This leads to a variety of emotions, reactions, interactions, and behavior changes. Attitudes toward other agents consist of an emotion and a correlation magnitude, and are subject to change during the simulation. Social behaviors are introduced as actions that specifically relate to other agents, like giving orders, asking for help, or insulting. Some of the variables that affect such actions are the number of characters involved, the objects of interaction and types of offers and outcomes. Given a social scenario, the agent chooses between a collection of social goals like helping, or making

friends. The concept of social norms is presented as a boundary or condition set to the character's behavior, and the emotional and practical consecuences of breaking or honoring them. For example, if the character feels sleepy but can't go to sleep since he is at a party, he will instantiate other behaviors or actions that will take him to bed (like excusing himself, standing up and leaving the room) and then the goal of sleeping can be accomplished. In this case, excusing himself is called a segue goal, or an intermediate goal between the current and the next one, so that an acceptable transition can be performed without breaking social rules.

Another project that relates to characters capable of engaging in social interaction is described by Rousseau[62]. In this architecture, all the conditions necessary to a simulation, like emotional predisposition between characters, or the mood of a particular character, are set *a priori*. The agent owns a collection of actions available to be performed when and if certain conditions are met. Each action relates to specific personality traits and by selecting them the overall attitude or personality of the character can be modeled. Before the simulation, the designer determines the characteristics of the episode including a description of the social situation, and number and personality of the characters. The inner emotional magnitudes and correlations of each character with respect to itself and others must be hand–coded before the simulation starts. At each cycle, the internal values of the character are adjusted and updated so that different actions can be proposed and ultimately chosen. Depending on the action, several expression methods with different mood characteristics are available. The character choses the one that best suits its current emotional state. For example, if the character is to walk from A to B and is currently *angry*, something like an *"angry–mood–walk"* will be chosen. This simple model has the advantage of making characters react in different ways depending on their social situation, their current mood and the character with whom they are interacting. For example, our character might walk in a different way when he is alone, with a small group of friends, and within a big group of people.

In Kline[34] we have an example of excellent integration between perception and cognition. Perception is done by physical sensors (like heat, proximity, pressure); synthetic vision (detecting objects, other agents, or features), and direct sensing by directly querying the world. Sensors filter the information coming from the world or other sensors. Each sensor reacts to a spectrum of data. The eyes, for example, sense an image that constitutes input to other sensors that detect individual objects, then animals, a later sensor identifies a dog, and so on. The work of transducers is to get the output from sensors and construct specific numerical information that can be interpreted by the inner rational systems. The whole architecture is based on accumulators, which are equivalent to drivers. Accumulators are organized into groups that behave in distinctive ways. Actions are generated by accumulators with positive or negative gain depending on the stimuli. Drivers send output directly to the action systems so that expression is performed. Some behaviors will inhibit or precede others. Personality is modeled by the way emotions are promoted, inhibited or cross–canceled.

Another project with interesting perception, learning, and cognition systems is found in Isla[16, 28]. In their architecture, called C4, the world is presented trough data records that might or might not be interpretable by

the characters. Each character has sensors for certain data records that, once recognized, are passed to the perception system, which transforms sensed data into meaningful information. Data is perceived through a set of *percepts* (anatomical classification and data extraction units) that analyze information in various dimensions (sound, shape, movement pattern...) to determine a matching probability with respect of known entities. If this probability exceeds a threshold we assume the object is known and more particular information can be retrieved. Percepts are hierarchically linked; *shape* percepts, for example, detect shapes and pass information to their children, who might specialize in more particular kinds of shapes. An internal representation of the world is maintained and used to extrapolate or complete information about sensed objects. This way, prediction and surprise can be generated. Learning is done by different operations over the perceived information and the results of performed actions, thus updating, refining and creating more internal structures to improve behavior. C4 provides an action system that evaluates aspects like what to do, when to do it, how worth is a certain action, among others. The user is presented with a 3D environment where the character performs a set of previously hand–made animations in response to its chosen actions.

Brooks presents interesting architectures for active and reactive behavior in real robots. In [9], an incremental set of augmented finite state machines (AFSM) connected to conventional finite state machines, provoke or inhibit the action of several motor systems that control the walking behavior of a robot. Messages from the different elements trigger changes of state and cause the robot to act. Although there is no semantic information in the sensed data, the reactions of this *physical* multilayered system are similar to the primitive behavior observed in animals. Along with Brooks, Lorigo and Grimson[38] describe another architecture for perception–expression behavior in which a robot with a vision–based system utilizes three different systems (Brightness, RGB color, and HSV color) to analyze and navigate unstructured environments. The robot creates a depth representation of the perceived scenario by merging output from the three modules, and transforms such representation into motor commands. In [8], Brooks and Stein propose the ambitious project of creating a system that would actually help to understand human cognition. Their approach is to use massively parallel machines to process the enormous amount of incoming stimuli from the world, effectively control more than one concurrent and/or cooperative processes (like moving the eye and the neck to face a target), and provide support to complex high–performance software on the humanoid model. A physical robot implementation would provide stereo vision, four–channel sound input, a pseudo–anthropomorphic body with kinematic sensors and actuators, among many other features. Intended cognitive abilities include several classic visual tasks like tracking and gesture recognition, and auditive tasks like voice/face association and natural language feedback.

A simple and accurate goal–oriented system is described by Rizzo[50]. In this architecture, personality is defined by the distribution of priorities and magnitudes among a specialized hierarchical structure of goals. A timid, selfish person, for example, will have a different hierarchy of goals than those of an extroverted and generous person. Goals are seen as activities like resource provision (supporting other agents while avoiding selfish behavior), material gain (increase amount of resources), belongingness (building relationships

of various kinds), image (maintain a good public image) and others. Plans are different strategies for pursuing goals. Goals and priorities are always tried to be satisfied, but for a particular goal there will be usually several available plans, all with side–effects. Such effects will affect other goals. Stealing, for example, is a plan for material gain but also affects image, so a certain character will decide not to steal, while another character with less priority for image will have no trouble stealing. Personality is modeled by task discrimination given the impact of proposed actions over the character's goals, and the activation of different tasks depending on the character's priorities. Another goal–oriented project that makes use of planning, and explores the relation between emotions and decision–making is found in Gratch[23, 24]. Graphs of cells containing preconditions or constraints, actions, and world state predictions are organized to form a particular plan. Plans can be extrapolated to predict the outcome of an action and compared with those of other characters to determine if they conflict or favor each other. Emotions, inspired from Ortony[3], come from the relationship between the character's goals and observed information or events. For example, if the goal of character A is buying candy, and the plan includes using his money, which was just taken away by character B, A might feel frustrated because the goal won't be accomplished, or angry with B for hindering his original plan. This project is oriented to model realistic human behavior in a high–end VR training environment for military personnel, and is among the very few that include an immersive environment as part of its architecture.

The work of André[2] is another example of goal–oriented systems. In this architecture, affective states can be generated and expressed to the user in predefined ways, for example, extrovert characters use stronger phrases and talk more often. There are two channels for affective information: primary emotions (physical, involuntary) generated by heuristics, and secondary emotions generated by affective reasoning. Emotions are used to determine an emotional state which is immediately reflected by the character's expression system, including physical reactions and biasing when choosing dialogs. Goals and motivations can be re–ranked depending on the situation. The character will always try to execute actions that take him to accomplish a particular goal. There might be more than one active goal and several ways to achieve it.

We find a good example of driver–generated behavior in Velázquez[71]. This is one of the few works that look for the generation of emotions in *real* robots. Here, the world is sensed via physical sensors like temperature, vision and tactile recognition systems. Perceived information is filtered and processed, affecting the magnitude of several drivers. Emotions are grouped in networks by families like *fear*, *anger* and *enjoyment*. Each emotion has a corresponding *releaser* that constantly looks for certain conditions in order to trigger a visible response. Like in many other projects, there are functions to control decay and magnitude of expressed emotions. Personality is modeled by controlling levels of arousal and cancellation between emotions, certain emotions can be inhibited, while others are promoted. Learning is done by an associative network that relates emotions with the stimuli that generated them and the amount of energy involved. If the agent finds a marked stimulus (something that has triggered an emotion before), the corresponding emotion is triggered again and detected by the releasers, creating a biased behavior. Drivers represent instincts or needs. Each driver has a set of releasers and when its value differs from the desired, a message is issued and a corresponding

behavior is generated.

Another interesting model of emotions is proposed by Nicholson[5]. Emotions are represented with a certain magnitude, considering a desired value as the maintenance value. The stability or instability of the character will be defined by its emotion variability. Emotions are put inside a network where each node contains both the emotion and a number of targets, this way any emotion can be directed to several other characters or oneself, and relations between emotions are determined by weighted arcs within the network. A changing emotion propagates through the network affecting other emotions. The system has several modules to control behavior. The physiology module, for example, consists on a series of drivers that represent physiologic needs. Some drivers depend on the cross–relation with other drivers (health, for example, depends on tiredness and hunger). Goals are created to satisfy the character's needs, both physiological and emotional. Actions derived from goals are triggered by priority, and those with higher priority will interrupt previous ones. Emotions influence behavior by enabling, disabling, and triggering actions. Enabling and disabling is done by defining preconditions requiring certain emotional and physical values. For example, eating might require that fear has a low value, therefore our mouse would not be able to eat while the cat is around. After executing an action, the system of drivers updates their values (after eating, hunger would decrease to normal levels). Agents can store opinions of other agents based on their first encounter and the interaction history. When 2 agents meet, their profiles and emotional states are correlated. A state–transition model allows consistent sequences of actions to be executed concurrently, and rejects those that need to wait for other conditions to be met.

2.3 Discussion about Related Projects

Oz and other related projects[31, 32, 57] demonstrate an excellent implementation of goal–based emotions and behavior as an alternative to driver–based approaches; it also hints on how to create a good interface between general emotions felt by the characters (like fear, anger) and the unique corresponding reaction of each character (behavioral features). Nevertheless, we must remember that not every behavior is goal–based, and factors like learning and intuition must be also integrated.

Rousseau[62] provides a model in which characters are easy to configure, the design of complex relations is straightforward and there is some variation depending on their personality. However, there is no *real* social cognition since behavior is based on rules, the simulation is quite deterministic, and the lack of learning, improvisation capabilities, and other high–level techniques makes the model unable to sustain a character in a real time immersive environment. This is completely understandable, since the author's goals and domain are not oriented for that purpose.

Kline[34] presents an excellent set of building blocks that can be used to create both simplistic and complex characters. There is good interaction between the different modules and the presented perception system (filtering information by means of content–specific filters) seems quite efficient. All

components show good modularity, which is an essential factor when making modifications to an already existing character. The model leaves learning and social interaction out of the picture, and seems to be merely *reactive*, probably appropriate for simple characters. The definition of feelings and emotions is also quite minimalistic, but undoubtly sufficient for the project objectives.

C4[28] is an amazing project, complete in almost every aspect. Excellent perception system by using hierarchically organized precepts; efficient and effective learning algorithms; interesting proposition for learning, prediction, and even surprise; good integration between perception, cognition, and expression. Presently it lacks a model of emotions, roles, and social cognition, and a more proactive action system.

Rizzo[50] presents a very simple and accurate system from a goal–based perspective, designing different personalities seems straightforward and the priority tree is an interesting idea for taxonomic representation of goals. Nevertheless, we find no variability between characters with similar goals and priorities, there is no real personality but a particular priority configuration. Characters are static in the sense that they will not change their internal structures from what they learn in the virtual world, their personality remains always the same. Also, there is no real influence from emotional states in the decision making process, just a set of guidelines (priorities and goals) for selecting the next action to perform.

The general advantage of goal–oriented and plan–oriented systems, including those by André[2] and Gratch[23] is that behavior results more proactive, reflecting the role and intention of the character. A strong disadvantage is that usually the same goal structure is used to determine emotions. We must remember that not all emotions are generated from the relationship between events and goals. The psychological characteristics of our synthetic characters should (hopefully) be more interesting than that. It has been proved by authors like Ramachandran[56]that emotions can be elicited even *before* the rational evaluation of an event takes place; in fact, a malfunction in the human emotional system may lead to an erratic rational perception. For example, a person with selective cerebral damage might lose the ability to elicit affective emotions in response to a close relative, therefore his relation with such relative is profoundly affected. This and other evidence (Capgras and Reboul–Lachaux, 1923; Ellis and Young, 1990; Hirstein and Ramachandran, 1997; Baron–Cohen, 1995; Trimble, 1992; Slater and Beard, 1963; Bruens, 1971; Bear and Fedio, 1977; Damasio, 1994; Goleman, 1995, Kort[36]) highlight the importance of emotions in cognitive processes, and suggest that in some situations, emotions are the creative force behind perception, interpretation, and planning, and not the opposite!

Brooks exposes perception–expression systems that result adequate for very specific robotic tasks. A good contribution to the field of synthetic characters is precisely the direct link between perception and reactive behavior. However the tasks he intends to achieve, as useful as they might be for the domain of real robots, are irrelevant and unpractical for our purposes. The same behavior demonstrated by an incremental array of finite state machines can be implemented with a simple set of rules in a synthetic character; the notion of a perception system that intends to stream *each and every* aspect of the

world into the character's internal systems is precisely the problem many architectures, including ANIMUS, pretends to avoid; and the customization level of the proposed perception systems is too rigid to fit conceptual or artistic design constraints.

Two common issues exist on most reviewed projects. The first one is that behavior is often reactive; with the exception of goal–based and plan–based models, characters are always waiting for something to happen, for conditions to be met, for rules to apply. The second issue is that emotions are often put into the model for display only. Emotions should serve as an influencing force for the character itself, biasing its complete internal structures and not just its way of walking or speaking. I believe Nicholson[5] gets closer than any other authors to this idea by actually favoring or hindering the character's array of options by their current emotional state. In any case, emotions should have an effective impact on the character's inner rational processes.

Since there is no universal solution for the cognitive process, we leave the cognitive layer of the ANIMUS framework as a propitious environment for the designer to implement the models and techniques that best fit each particular character. For this purpose there are several structures (including drivers, specialized memory modules, messaging systems and flexible perception and expression integration) that facilitate the adaptation or creation of techniques as simple or complex as required.

Project	Focus on	Features	Application
Oz[44]	Expression	Goal–based, Emotional states	Interactive drama
Reilly[57]	cognition, expression	social behavior and behavioral features	drama, social interaction
Rousseau[62]	expression	*a–priori* conditions, hand–coded and rule systems	drama
Kline[34]	general	driver–based, cross–arousal and cancellation	entertainment/3D
Isla[16, 28]	general	memory, learning, automated segmentation, conditional triggering, high–level simulation	entertainment
Gratch[23]	cognition, expression	conditional triggering, weight–based, goal–based, high–level simulation	training
Rizzo[50]	cognition	goal–based, priorities, behavior agenda	simulation
Andre[2]	expression	hand–coded, goal–oriented, emotional state	graphic/2D
Velázquez[71]	expression	driver–based, cross–arousal and cancellation, physical sensors, conditional triggering	real robots
Brooks[9, 8]	perception, expression	finite state machines, augmented state machines	real robots
Nicholson[5]	cognition, expression	driver–based, cross–arousal and cancellation, emotional states, priorities	simulation

Table 2.3: Related projects and their general features.

Chapter 3

Basic Concepts and Definitions

"Omnium rerum principia parva sunt"

-Marcus Tullius Cicero

Now that we have reviewed some related projects, we describe some general concepts that will be useful to understand the ANIMUS framework. We also illustrate several configurations for setting up a virtual environment, and the nature of information interchange between the ANIMUS synthetic characters and the human audience.

3.1 Definitions

Many of the concepts utilized in this thesis are not specific to this work. They are widely used by people and groups conducting similar research. Nevertheless, some of the terms will imply additional assumptions, while others are completely adapted to our own research. To avoid ambiguity we define these terms as they are to be understood within the current framework. The following definitions reflect only their application to this work.

A *character* is an abstract concept that denotes a unique imaginary creature, human or otherwise, along with its/his/her individual attributes, traits, roles and personality. Characters *per se* begin as an idea in the mind of their creator(s). They can be represented by a human actor in a theatrical play or movie, brought to life in animated films or narrated in books; in any case, they are put into existence by means of different forms of art.

A *virtual character*, or synthetic character, is the representation of an abstract character inside a virtual world. It is important to note that a virtual character must appear as believable as a character in any other artistic representation. A walking 3D spider in the virtual reality environment is not necesarely a character *unless* it performs a role, interacts with other creatures and the user,

and demonstrates its personality. Similarly, an animated anthropomorphic avatar that mimics physical needs like hunger and instinctively looks for food in a 3D domain, looking happy when fed, may be a good virtual creature, *but it is not a synthetic character*. It is necessary to express goals, emotions, a clearly defined role and personality. As Thomas and Johnston state[68]:

> "Consider this: a strong situation has been established. The character comes into it with a definite and interesting attitude. Confronted with the problem, he develops his own personality, grows a little, shows who he is, makes his decision for action, then does it in an entertaining way" (p. 375)

While an actor represents a character in real life, a *synthetic creature* gives life to a character in the virtual world. A synthetic creature is merely the *virtual body* in the immersive environment that brings the character to the eyes of the audience. Such body will be as good as its ability to express the nature and personality of the character it represents. A synthetic character will fail if it cannot convince the public of its authenticity.

An *entity* is a very interesting concept. In principle it denotes any synthetic character *regardless* of subjective role and personality interpretation. An entity is the part of a character that everyone perceives in the same way. For example, imagine you are dining with a group of friends at a restaurant when a cat enters, walks to an empty table and meows to the bartender, who immediately pours some whiskey into a cup and takes it to the cat's table. The cat drinks the whiskey, yawns and walks away. Now each one of your friends might be wondering what kind of character they saw, discussing its human–like attitude reinforced by the reaction of the bartender. At the same time everyone perceived exactly the same entity: the visual and auditive information of the cat, stripped of all interpretation. The nature of the cat's personality can be debated, but the cat as an entity remains the same for everyone. This distinction is useful because it allows different characters to essentially perceive the same entity, but have a unique concept of such entity given their personal interaction and experience. This will solve ambiguity when refering to information at different levels. The perception layer[1] works with *entities*, while high–level cognition systems can recognize *unique characters*.

An *agent* is a semi–autonomous and proactive program specialized in performing some particular task. A better definition is provided by Hayes–Roth[25]: *"Intelligent agents continuously perform three functions: perception of dynamic conditions in the environment; action to affect conditions in the environment; and reasoning to interpret perceptions, solve problems, draw inferences, and determine actions"*. In this project we will find several types of agents at different levels of the architecture.

From Torres and Boulanger[69] we define the concept of *feature*: A feature is the basic information primitive in our architecture. It consists on a concept–value pair similar to what is proposed in Bates[31], and represents anything that can be sensed by a synthetic character. Features range from physical properties like the temperature, color and weight of a table, to

[1] As explained in Section 5.

complex concepts like pain, hunger and fatigue. If a feature is produced by a character and detected by another we call it a *stimulus*. It follows that *stimuli* is the set of features expressed by a character at a given time. As a consequence to a stimuli, the virtual characters will generate a response that constitute stimuli to other characters and the human audience, generating a cycle that remains for the duration of the immersive experience. The magnitude of a feature is not limited to a particular range, but must be congruent to the concept it represents.

An important concept is that of *sensory honesty*. In order to preserve the believability of our characters, their perception must be bounded to the physical and logical conditions of each situation and their own constitution. For example, a blind character must not be able to react to visual stimuli; a deaf character should not startle if someone shouts; If character a stands behind character b, then b must not be able to see a unless it turns around; the voice volume of a should decay as b moves away; a cat should not see the mouse if it hides behind a wall. Stimuli is rarely perceived by some creature as it was expressed by another; it is often adapted to make it relative to each affected character. Characters must be always prepared to deal with noisy or missing information.

An *immersive environment* or virtual reality world is the scenario where our virtual characters live. It consists on a 3D environment that is presented to the user by means of visual and sensible feedback (graphics, sound, haptic rendering...). The audience produce stimuli by using sensors that detect their position, movement, sound, respiration or any other available or adequate source of information. Stimuli from the user are processed in the same way as stimuli from the characters, and so humans and virtual characters are able to interact with each other.

3.2 Setting Up a Virtual World

The complexity of the virtual world setup will depend on the nature of each project. For a common videogame a simple monitor and a keyboard or joystick will suffice, while an artistic installation or more advanced videogames might require more than one projector for creating stereo images, specialized sensors depending on how we want to interact with the characters, and any other immersive or interactive device adequate for each circumstance. In all cases, the ANIMUS framework remains as an independent component; it can be easily adapted to assimilate information coming from any kind of input (be it a joystick or a galvanic skin resistance sensor), leaving the rendering task to agents specialized on each particular setup or equipment.

Figure 3.1 illustrates the abstract layout of a common immersive environment. Maintaining the virtual world and its characters is the responsibility of the ANIMUS framework. Interaction and feedback from/to the user will depend on the flavor of the VR setup we decide to implement, as defined for example by an InTml file as in the works of Figueroa, *et al.*[49]. Basically any kind of sensor can work as an input device. For example, if a character is supposed to be able to "feel" the user's emotional state, input from galvanic skin resistance, breathing, and heart rate sensors can be mapped into appropriate

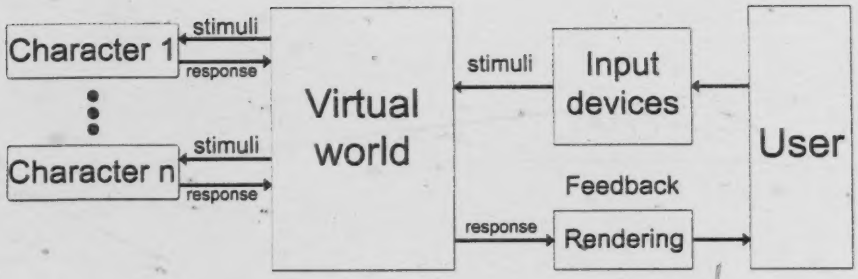


Figure 3.1: Abstract layout of the immersive virtual environment.

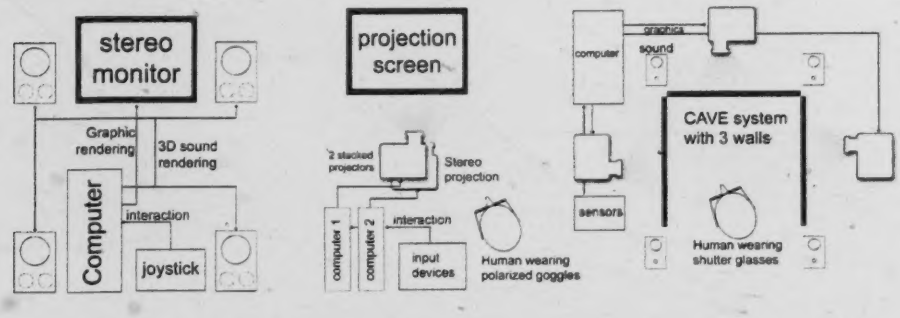


Figure 3.2: Several configurations for a virtual reality environment.

features perceptible to the character, which in turn should react to them according to its role and personality. The rendering or feedback module contains a set of agents that effectively present the world and its characters to the user. These agents vary depending on the chosen display devices, graphic library, haptic hardware and such. Rendering tasks are not limited to graphics and sound, they may also include haptics and other forms of specialized feedback. In Figure 3.2 one can see several options for setting a virtual reality environment. Probably the most common and economic (and also less immersive) is that of a videogame, where feedback comes from a conventional computer monitor or tv and sound card, and interaction is provided by a joystick. Other setups, like a single–wall stereo projection, offer a more interesting experience but also require specialized or expensive hardware. It is up to the conceptual artists and designers to choose a virtual reality setup that satisfies the needs and nature of their project.

3.3 The Flow of Information in the ANIMUS Framework

Figure 3.1 also illustrates the basic flow of information in the virtual environment. There are five elements in this diagram: The user, input devices, rendering devices, the virtual world manager, and one or more ANIMUS characters. The central element of this architecture is the virtual world manager, which is inspired by Isla[16]. Its four main functions are:

1. To maintain a list of characters present in the VR environment at any time.
2. To facilitate the interchange of information and stimuli between virtual characters, preserving sensory honesty.
3. To synchronize important events like perception, expression and animation frequency.
4. To convey all necessary information to the rendering agents and read the stimuli created by the user from the input devices, so that virtual characters can be aware of it.

The virtual world manager has a specialized *blackboard* that keeps a record of all stimuli produced both by the virtual characters and the human audience. Stimuli from the characters is generated by themselves, while stimuli from the user is collected from the available input devices. When characters update their perception systems, they *sense* other characters and the user[2] by reading the virtual world manager's blackboard. All information is transformed by the manager *before* it is retrieved in order to make it relative to each character and preserve the sensory honesty principle. For example, if character a wants to know where b is standing and it happens that b is occluded by a wall, a will not be able to know b's position, even if b has published it as a feature. Characters can directly query other characters for particular features, but since information goes through the same transformation process, every character *must* assume that the response –like any perceived stimuli–, might be noisy or absent at all. Characters are not obligated to publish their internal features, nor to give them away when somebody else asks. If a is looking for b but does not find its position in the blackboard, it might try to directly ask. It is understandable that b will never answer such call if its intention is to remain hidden. Human audience perceives the virtual world and its characters through feedback systems that include but are not limited to graphic and sound rendering, as mentioned in Section 3.2.

ANIMUS provides tools for simplifying the overwhelming task of managing information and knowledge from the world, so that designers and artists can focus on the implementation of the character's behavior at a higher level of abstraction. Part II provides more details on how such tools can be applied.

The interchange of information to and from each character is synchronized with the world by a sequence of events signaled by the world manager itself.

[2]It is interesting to note that, for any given character, the user also *looks* like a character.

These events prompt the characters to update their perception layers and publish their own stimuli in the blackboard. Table 3.3 shows the different events and the character activity. Figure 3.3 shows the interaction between the *perceive* and *publish* phases. The *Information adequation* section transforms a stimuli to make it relative to the perceiving character. Once a character is registered in the world manager, it will be automatically queued for these events. Remember that characters are not strictly required to perceive or publish information.

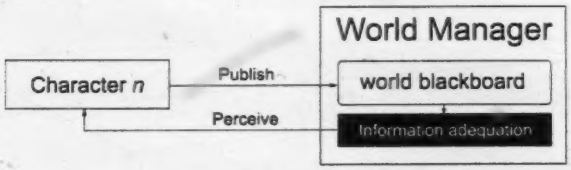


Figure 3.3: Perception–related events triggered by the world manager.

3.4 Summary

In this chapter, basic concepts have been defined as they apply to the current framework. We stated the important difference between a 3D *creature* with or without intelligent behavior, and a *character*, which integrates a unique role and personality.

We outlined the abstract layout of our virtual environment, its components, their role, and the nature of their interaction. We discussed how several factors, such as the requirements of the project and the type of characters we design, affect the complexity and functionality of a virtual reality setup. We provided several examples to illustrate some typical setups.

The virtual world manager was introduced and its functionality explained. We described how virtual characters produce stimuli and perceive that of other characters by means of a specialized blackboard. We denoted how this blackboard preserves the principle of sensory honesty by making perceived information relative to each character. Stimuli from the user are also included in the blackboard, and specialized rendering agents provide several types of feedback (essentially graphics and sound) to the user.

Table 3.1: Events triggered by the world manager.

Event	Character Activity	Frequency
Perceive	Characters are given an opportunity to read the blackboard and update their perception system. In other words, an ANIMUS character is able to collect information through its available input channels. The system guarantees that information will remain static until the next perception phase (All characters perceive the same world state)	aprox. 25 times per second
Publish	Characters are asked to *publish* a set of features related to their current state. The world manager creates data records for each character and their information in a fashion similar to Isla[16]. This information is kept on the world manager's blackboard.	aprox. 25 times per second
Animate	This event ticks the expression layer to render the character on the available output displays. This is not limited to graphic interfaces, it may include haptic and other specialized devices.	Depends on hardware capabilities. Aprox. 60 fps

Part II

The ANIMUS Framework

Chapter 4

Architecture

"You know I always thought unicorns were fabulous creatures too, although I never saw one alive before."
"Well, now that we have met," said the unicorn, "If you'll believe in me, I'll believe in you."

-Lewis Carol, "Throught The Looking Glass"

Let us propose a question to the reader: *How do we create life?* The genesis of a new, self–sustained, autonomous, intelligent, living organism. Perhaps the question is not adequate: creating life is a gigantic task of divine proportions. Let us suggest a different question: *Can we create life at all?* At this point our interviews with scientist, philosophers, biologists, mathematicians, doctors, sociologists, writers, dramatists, poets, and other fellow humans of important and distinguished professions fill our notes with opinions that –I suspect– result almost as chaotic and complex as life itself. This question is difficult because life can be seen from many subjective perspectives, we do not know what life is, or at least there is no simple definition that peacefully fits, for example, a scientist, a theologist, and my friend the mathematician. I remember a conversation with a physicist who stated that life is the set of rules, principles and methodologies (known or unknown) that govern the universe, therefore the problem is not if we can create life or not, but how can we control and understand it. A biologist gave us a short talk about DNA, the last cloning research, genetic engineering, and how in the future it will be possible not only to create life, but to *program* it to perform in particular and desired ways. A priest asked me to abandon such heretic intentions, urging me to repent and brace life as a way to salvation, while another priest solved the matter by stating that we simply cannot respond to that question: creating life goes beyond man's possibilities.

Nevertheless there *was* someone who said that we can, indeed, create life. "The best example" –he said, "is Tom Sawyer". I wondered why he claimed that Tom Sawyer was alive. "It is simple: Ask me who Tom Sawyer is, and I will tell you his story; at the end you will wonder if Tom Sawyer was real or not, in the same way you wonder if certain chapters about Napoleon's history

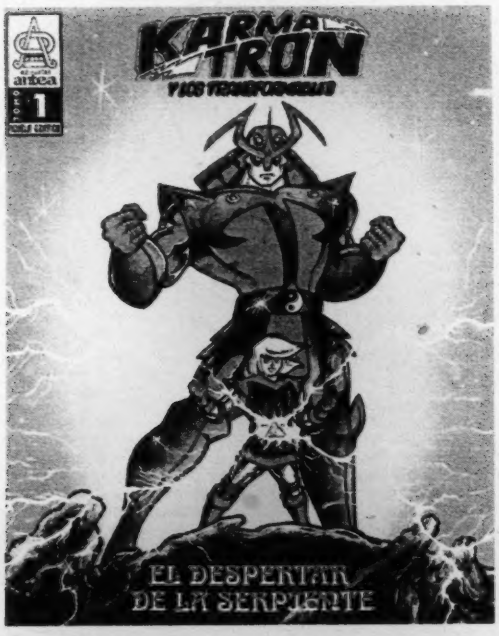


Figure 4.1: Karmatron y los Transformables

are fantasy or fiction. But because you wonder, and you can imagine how Tom Sawyer would look and act today, I can assure you he was alive and the only argument against me would be Samuel[1] claiming that he was guilty of making everything up". Needless to say, I was amazed by this argument, so I began to ask people about fantastic characters from novels or fiction stories that they could easily imagine *alive*. To my greater amazement, many of these characters were not even human! "Why do you say that *Karmatron*[2][40] (Figure 4.1) could actually be a *real* person? how can you imagine someone like *him* alive?" I asked one of my subjects. "Oh, of course I know he is fictitious" –she answered, "but the story is so richly narrated, and the character is so perfectly described that I just have no problem imaging him alive". Other examples included characters like *Snoopy* and *Mickey Mouse*. In all cases, it was clear that people were well aware that such characters were fictitious, **but they would voluntarily suspend their disbelief and imagine them alive!** There were, of course, other imaginary creatures that according to them *did not appear as real as other imaginary creatures*. What was the difference? What makes a fictitious character appear to be alive?

At the same time, I asked their opinion about computer programs like *Deep Blue*[3], *Alice*[26], different videogame characters, graphic character representations like those of `h|anim`[27], *lifelike* simulations like Funge's[30] merman and Terzopoulo's[13] swimming fishes. Most people agreed that those were indeed

[1]Samuel Langhorne Clemens is also known as Mark Twain.

[2]A Mexican superhero

[3]©IBM Research

interesting *programs*, but none appeared truly alive for them; not even the ones that were supposed to closely mimic their real counterparts. "Of course those fishes swim like the real ones" –commented someone, "but what else is interesting about them?". Probably such projects would be interesting for people in the field, I personally find IBM's deep blue and Terzopoulos's fishes amazing, from a technological point of view at least.

It seems that while scientists are able to create *intelligent agents*, artists are able to *imagine life*. The problem seems to be that artists have difficulty creating *interactive life*, and scientists ignore the artistic requirements of an *authentic* fictitious creature. As mentioned in Section 1.3, art and science should integrate their knowledge to create successful *synthetic characters*. We now return to the question: Can we create life? my intuition is: We might not be able to create life, but we certainly can create characters that *seem* alive; artists know how to conceive them and scientists have the knowledge and technology for making them interactive. This architecture is an attempt for such integration. It proposes to put Art in the leading role, guiding the hand of science for the creation of a new generation of interactive synthetic characters for immersive environments.

4.1 A Three–Layer Architecture

The design process for a synthetic character using the ANIMUS framework begins with the conceptual artistic design. A solid and consistent character design must be able to respond to the following three questions:

1. **How does the character perceive the world? How does it instinctively react to the world?**
   What perception channels are available to the character? Is it able to hear and see? how accurate? Probably the character is completely blind, but its hearing sense is highly accurate. How does the character's instincts affect its behavior? Possibly the character has the ability of a cat: excellent sense of balance and very accurate and precise movements. Maybe the character is a natural enemy of light, always lurking in the shadows and turning away from any bright object. The *Perception Layer* allows the artist to determine the basic physiological perceptive and reactive nature of the character. This will depend on what kind of *creature* or character is, and will greatly influence its behavior. The perception layer is explained in Chapter 5.

2. **What happens *inside* the character?**
   What is the *unique* personality, role, intention, psychological and emotional constitution of the character? What does he likes? What does he hate? Does he have any social behavior? Can he make friends? Probably he is more like an animal and his reactions are simple and deterministic. He might be an evil genius and his actions are intelligent, carefully calculated, he hides his feelings to conceal his intentions. He could be a psychopath, always behaving in unpredictable ways, or a friendly and patient character. What are his goals? How does he make decisions? Is he intelligent or is he more or less a fool? Is his behavior funny, sinister, selfish, sincere? Is he open minded? The *Cognition Layer* allows the

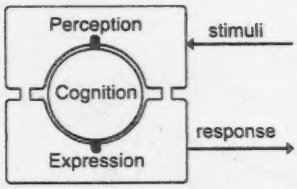


Figure 4.2: The three main components of an ANIMUS character.

designer to create more or less complex systems that determine the internal high–level cognition of the character. Such operations might range from simple rule–based systems, to novel cognitive, knowledge–based, AI techniques. The cognition layer is explained in Chapter 6.

3. **How does the character express its personality and internal emotions?** How does the character move? Is it heavy? Is it agile? How does it walk? How does it wave its hand? Does it has a particular way of looking or scratching its nose? Is it a martial–arts expert and its movements demonstrate its expertise? Or is it clumsy and rather funny to watch? Does it look proud or pity? Does it look to the floor when nervous or smile when happy? The *Expression Layer* allows the artist to determine the expressive nature of the character; its unique way of acting; its original style of movement. This aspect is of the utmost importance, since non–verbal information is crucial for the believability of the character. Animated films, for example, heavily rely on expression to convey the personality of the character to the audience. Through the character's animation we must be able to guess its thoughts, emotions, intention. Chapter 7 describes this layer.

The ANIMUS framework is designed in such way that responding to these questions determines the character's internal configuration and functionality. After the artist has created a detailed profile of his/her character, the designer takes such profile to develop several internal structures that take the creature to behave as specified. The perception layer, for example, will implement input channels of adequate type and quality, and the configuration of reflexes that conform to the character's nature. The perception layer will include the appropriate AI, decision making, emotional models, or high–level methodologies to implement the character's desired behavior. The expression layer will give the character the ability and flexibility to control his own animation, constrained by the unique style and dramatic concept created for him by the artist.

The three components of the ANIMUS framework can be seen in Figure 4.2. One of the main differences between ANIMUS and other architectures is its modularity and interconnection across layers. Stimuli from sensors in the immersive environment, other creatures, and world objects, enter the character through the *perception layer*. The perception layer is directly connected to both the cognition and the expression layers. This interconnection allows the character to trigger automatic reflexes even before its brain recognizes the perceived stimuli. This process is similar in biological organisms: If I am peacefully reading a book when the fire alarm goes on, my whole body

instinctively reacts to the sound (I might jump in surprise, or turn my head to the origin of the noise) a fraction of a second *before* my brain thinks "Ah, the fire alarm! I should get out of the building". This provides a faster, automatic reflex similar to the ability of a fly when evading incoming objects (like a slapping hand) because its perception system is directly linked to its body (it does not need to *think* what to do to get out of danger). The cognition layer has access to both the perception and expression layers, so after hearing the alarm and jumping with surprise, a virtual character might decide to stay calm and head out of the building. The cognition layer can query the perception layer to ask questions like "Where do I see the nearest exit?" "Is there fire inside my room?", "Do I see someone else besides me?". After that, it can direct the expression layer to execute motor operations to get the character out of danger. Visible feedback to the audience comes from the expression layer.

4.2 Some Claims about ANIMUS

Before describing the ANIMUS architecture in detail, it is important to make certain claims to avoid misunderstandings about the capabilities and nature of this project.

1. ANIMUS does **not** look to create AI algorithms that perform in the most efficient or optimized way. Our intention is not to design the fastest, stronger algorithms, but to facilitate the creation of *interesting* characters. Algorithmic research is an exciting, but different area.

2. We do not claim that ANIMUS characters are *intelligent*. Sometimes intelligence is not desired; and when it is, the responsibility of implementing intelligent behavior rests on the hands of the designer.

3. ANIMUS, like any other framework, is *not* an universal solution, nor is it the only one, *ergo* it is not necessarily the best for all cases. Given a certain character and domain, there might be better frameworks or methodologies. We only claim that this is *a* framework designed to create synthetic characters that comply to artistic definitions that would be difficult to achieve with other methodologies like traditional AI.

4. ANIMUS characters are not necessarily interesting or believable. That challenge is up to the artists and the designers using the framework. *Giving a brush to someone does not necessarily make an artist out of him/her.*

5. Currently this framework, particularly the cognition layer, **does not** implement any model of emotions or personality. It only provides tools to facilitate that task to the designer. Future versions will include configurable personality templates and models of emotions, but those remain outside the scope of this thesis.

4.3 Playing with Alebrije: A Practical Example

In this thesis, we will describe the creation process of an ANIMUS character through the use of an example project named *Alebrije*. Alebrije is a term used by some Mexican artisans to describe a particular kind of imaginary creature. In our case, Alebrije (a simple animal–like character) will serve to demonstrate the implementation and functionality of the perception and expression layers, and the basic principles of the cognition layer.

The core behavior of Alebrije is based upon a simple set of rules in the cognition layer. We have chosen to do it that way in order to maintain the example's simplicity, and because illustrating the model of emotional behavior *per se* is not the intention of this thesis.

We will begin by describing the artistic idea behind Alebrije in very general terms. Then we will proceed to the implementation details for each layer at their corresponding chapters. For a more in–depth explanation of an essential ANIMUS class, please refer to Chapter 8 and Appendix C. The reader will notice that the corresponding code for each layer only provides an insight on how some of the features can be implemented, but not the actual (i.e., complete) code that corresponds to such implementation. This was done to maintain simplicity and readability. Complete source code is available should the reader requires to see it as an example.

Another important goal of Alebrije is to demonstrate that a simple set of rules is enough to create realistic primitive (animal–like) behavior. Many research projects like Terzopoulos[13] and Funge[30] propose complex AI methodologies to produce similar results. Although such behavior is indeed achieved, the amount of computational overhead required could be saved for other demanding purposes like 3D rendering which, in real–life domains like videogames or visualization applications, tends to be very demanding. Contrary to common belief, AI processing and not realtime graphic rendering is –in many cases– the cause that multimedia applications run slower than expected.

4.3.1 A Brief Conceptual Design About Alebrije

Alebrije is a small lizard–like animal. He inherits most of the characteristics of a conventional lizard, particularly its agility and awareness. Alebrije is able to quickly detect small changes in the environment via his good eyesight and hearing senses, and his accurate reflexes make him a good hunter on short distance ranges. Like most animals, Alebrije is not capable of hunting more than one pray at a time, so when many options are available, he likes to choose and concentrate on one. Alebrije can get easily distracted and will find it difficult to concentrate if more than one event is happening at the same time. Visually, he looks just like a lizard, except for his skin, which is decorated by a pattern of antique Mexican–like runes. Unlike most lizards, Alebrije is not a shy creature and will not easily run away from an interesting stimulus. Still, he has the ability to change the color of his skin as an automatic response to the environment. In spite of his curious na-

ture, Alebrije will go to sleep if there is nothing interesting to pay attention to.

Alebrije has to look very agile, extremely aware of what happens in the environment. He must not be able to create intelligent hunting strategies, as his behavior is more impulsive and instinct–driven. One of the most strongly denoted characteristics must be his interest in perceived stimuli, although such interest is lost once he becomes used to existing phenomena. In other words, Alebrije must be able to distinguish known from unknown and show his eagerness to explore. For an illustration on the 3D body of Alebrije please refer to Figure 7.4

Chapter 5

Perception

"Fortune is arranging matters for us better than we could have shaped our desires ourselves, for look there, friend Sancho Panza, where thirty or more monstrous giants present themselves, all of whom I mean to engage in battle and slay, and with whose spoils we shall begin to make our fortunes..."

-Miguel de Cervantes, "Don Quixote"

Li occhi miei ch´a mirare eran contenti
per veder novitadi onde son vaghi
volgendosi ver´ lui non furon lenti.

-Dante Aligueri, "Divina Commedia"

The perception layer represents *reality* to the character in the same way our own senses enable us to interpret the world. A well designed perception layer is the basis for an effective and believable synthetic character. Perception plays a very important role in behavior and personality. As medical research proves (See Ramachandran[56], for example), a small defect in our perception system can affect our behavior, even when the rest of our mental faculties remain intact. For example, a person with selective brain damage is left with a paralyzed arm. The brain, having no feedback from such arm might induce a *neglecting* state, where the person assures she is completely normal and can actually move her arm. When asked to raise both hands, she could argue that she feels tired or that she does not like to follow orders. One might think that she is consciously lying, but in fact she really believes what she says. If the doctor directs her attention to the paralyzed arm she would even say "Oh, that arm is not mine, it belongs to my brother"[56]. Without going that far, one can discover differences in the behavior of a person with normal eyesight and a blind person, or the reaction of someone who tries a very spicy dish for the first time. In all cases, perception is our window to the world. Our behavior depends on what we see through that window. This is no exception

for our synthetic characters: a character with fine hearing capabilities will react sooner to sound stimulus than other characters; a character that relies on vision for navigation tasks will show itself disoriented if suddenly lights go off.

The ANIMUS framework implements an *early selective attention and perception system* that performs the following tasks:

1. It serves as an entry–point for the world, enabling the character to be aware of its environment within the bounds of its perceptual definitions. This means that a deaf character must not react to sound, a blind character will not detect light, a dog–like character will have better sense of smell than a human character, and so on.
2. It segments, discriminates and prioritizes information to bias the attention of the character toward salient stimuli *without* having to understand the meaning or nature of the perceived information.
3. It implements reflexes as an *automatic* response to certain stimuli
4. It performs low–level conditioned learning.

It is important to remember that the constitution of the perception layer greatly favors or harms the personality and believability of the character. For example, a blind character that notices the user before he makes any sound would not be believable. One must refrain from giving characters perceptive abilities they must not have, and we must correctly determine the quality of the ones they possess.

In the following section we will describe the architecture of the ANIMUS perception layer in more detail.

5.1 An Early Perception and Attention System

Most architectures perform semantic analysis of information as data is perceived (See for example, Isla[16], Rousseau[62], Rizzo[50], Velázquez[71] and Bates[31]) in order to generate a response. Other authors, like Brooks[9, 8], require enormous amounts of resources to process every possible perceptible stimulus in order to produce specialized primitive reactive behavior. Our architecture proposes a method for prioritizing perceived information *before* any cognitive or semantic analysis is applied. This presents three major advantages: First, because no complex AI algorithms are involved in the attention shifting mechanisms, it is possible to implement *instinctive behavior* and *automatic reflex responses* in a simple way, which greatly adds to the believability and realism of the character. If the creature touches a very hot surface, for example, it moves its hand away before having to decide that burning its hand is a bad thing. Similarly, bringing a strong source of light to its face might cause it to turn away or close its eyes. Second, it optimizes subsequent analysis of information by biasing the creature's attention toward

elements in the world that seem to be interesting, for example, a loud noise or an object that suddenly begins to move will call the attention of the creature, probably making it turn to face the source of distraction and ignoring static elements in the scene. Third, it allows the integration of low–level *conditioned* learning. Examples: a character that burnt its hand when touching a hot surface will instinctively avoid touching it again; a character that is always bothered by other character will begin to feel uneasy as soon as this other character appears, a virtual animal can be trained to respond in certain ways to some stimuli, while an identical animal that did not go through the same reinforcement training will remain unresponsive to the same stimuli.

Without information discrimination a character would have to interpret all perceived stimuli, which is inefficient in terms of resource requirements and performance. Human beings seldom perceive *every* detail of a scene; instead, attention is focused on particular elements[56, 45]. The possibility of discriminating information without having to analyze its meaning was inspired from the concepts of *early* and *late* selection on the perceptual process of human beings[45]. Early selection suggests that attention is directed to interesting stimuli *before* the perceived information is understood. This is done by blocking stimuli with certain characteristics, while preserving other. Late selection states that attention comes from the correct evaluation and interpretation of the perceived information by high–level cognition systems, in other words, we consciously decide what we pay attention to. It was later proposed that early selection would in fact perform *a priori* discrimination of the information, not by blocking, but by degrading or attenuating stimuli before it would reach higher levels of cognition. This means that our perception system favors certain stimuli and inhibits other, effectively biasing our attention mechanisms. ANIMUS creatures can get distracted by, or center their attention to the various events of an immersive environment instinctively because it is imprinted on the constitution of their perception layer, which reflects their artistic and conceptual design.

Definition	**Concept**	**Range**
i, j, l, n	Counters	variable
t	Time	variable
N	Total number of features in the system	variable
x_i	A feature or stimulus i	$1 \leq i \leq N$, $\|x_i\|$ is user–defined
M	Total number of characters in the virtual environment	variable
C_i	A character in the virtual environment	$1 \leq i \leq M$
S	The stimuli set of n stimulus	$1 \leq n \leq$ inf
s	A stimulus set containing j unique features	$1 \leq j \leq N$

Table 5.1: General ANIMUS constants and variables.

The architecture of the perception layer is illustrated in Figure 5.1. At a given time t, the world blackboard contains the stimuli set $S(t) = \{s_{1_t}, ..., s_{n_t}\}$, where $1 \leq n \leq M$, that is, one feature set for each character that produces

Definition	Concept	Range
a_i	Attention neuron sensible to stimulus x_i	$1 \leq i \leq N$
w_i	Input weight for attention neuron a_i	$0 \leq w \leq 1$
θ_i	Threshold for attention neuron a_i	$0 \leq \theta_i \leq \max(\lvert x_i \rvert)$
v_i	Net input to neuron a_i	$0 \leq v_i \leq \max(\lvert x_i \rvert)$
I_i	Output magnitude of neuron a_i	$0 \leq I \leq 1$
η_i	Learning rate of neuron a_i	$0 \leq \eta \leq 1$
δ_i	Decay factor for neuron a_i in frames	user–defined
γ	Decay function for the attention neurons	n/a

Table 5.2: Attention network constants and variables.

Definition	Concept	Range
ρ_i	Perception rate for character C_i	$1 \leq \rho \leq 25$
ω_{ij}	Variation factor for stimulus x_j perceivable by character C_i	$1 \leq \omega \leq \rho_i$
ψ_{ij}	Frequency factor for stimulus x_j belonging to character C_i	$1 \leq \psi\omega_{ij}$

Table 5.3: Variation module constants and variables.

an output. A given character $C_i, 1 \leq i \leq M$ reads the stimuli set $S_i'(t) \in S(t)$ such that $s_{i_t} \notin S_i'(t)$, this becomes the input to its perception layer. The output will be a prioritized stimuli set $S_i''(t)$ based on the results produced by the perception system. At the same time, the layer will perform all necessary low–level learning and reflex triggering, activating the character's instinctive behavior. The set $S_i''(t)$ will be used by the cognition layer to perform all required high–level processing.

5.2 Perceiving Intensity

Intensity of perceived information is evaluated on the *attention network*. The functionality of this module is similar to Koch[35]. A neuron a_j inside the attention network responds to a unique stimulus x_j and contains a temporal input weight $w_j(t)$ such that $0 \leq w \leq 1$, determining the sensibility of a_j to j, and a temporal threshold $\theta_j(t)$ that is initially set to 0 and represents the *attention umbral* of the neuron. In other words, if the character is able to perceive sound, there will be a neuron sensitive to sound stimulus that will get excited every time the character *hears* something. The weight of a neuron attenuates its input stimulus, and the threshold is a reference value that determines the magnitude strength. When a stimuli enters the network, the net

RA_{il}	The lth Reflex agent of character C_i	$0 \leq l \leq N$
$f_l(s)$	A reflex function belonging to agent RA_{il} over stimuli s	$0 \leq l \leq N$
ART_{il}	An ART network to synthesize stimuli that elicits a reflex from RA_{il}	$0 \leq l \leq N$

Table 5.4: Reflex agents and ART network constants and variables.

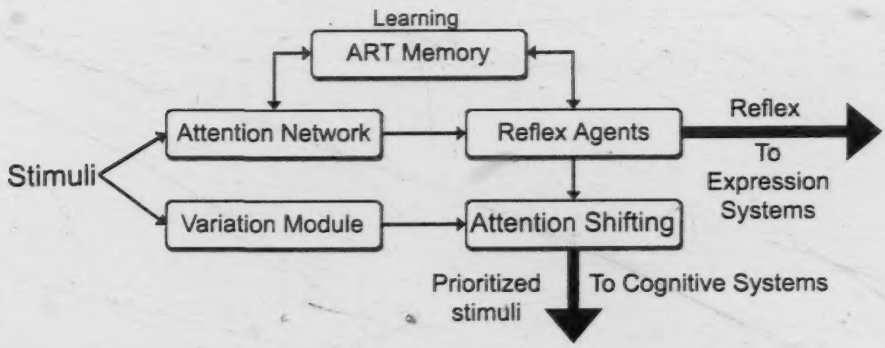


Figure 5.1: Architecture of the Perception Layer

input for a matching neuron is $v_j^i(t) = |x_j^i(t)|w_j(t)$, where $|x_j^i(t)|$ is the magnitude of stimulus x_j perceived at time t from stimuli s_i belonging to character C_i, and the output is an excitation level determined by $I_{j_t}^i \left[v_j^i(t), \theta_j(t)\right]$ (usually a sigmoid function) if $v_j^i(t) \leq \theta_j(t)$ and $I_{j_t}^i = 1$ otherwise. There are features that can only be produced as a combination of other features (like pain, for example), thus the excitation output of some neurons is linked to the input of other neurons creating the network. Both the input weight and the threshold of each individual neuron changes as a function of entrant stimuli and time, giving the neuron short–term memory characteristics. The value for $\theta_j(t)$ is determined as follows:

$$\theta_j(t) = \begin{cases} \max_k(x_j^k(t)) \, \forall x_j \in S'(t) & \text{if } \max_k(x_j^k(t)) > \theta_j(t-1) \\ \theta_j(t-1) + \eta_j(\max_k(x_j^k(t)) - \theta_j(t-1)) & \text{otherwise} \end{cases} \tag{5.1}$$

Where η_j is a learning rate value that controls how fast the threshold adapts to the magnitude of the stimulus, and a function $\gamma(w_j(t), \delta_j)$, usually a parabolic or sigmoid decay, takes the input weight $w_j(t)$ to converge to 0 in δ_j frames if the input remains equal or less than the current threshold. In other words, the character's senses adapt to the magnitude of a constant stimulus, which will cease to elicit attention after a certain time.

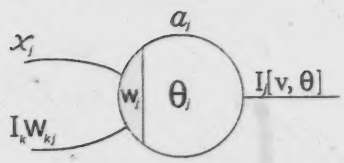


Figure 5.2: Structure of an Attention Neuron

The output of the attention network is a the vector containing the output of all excited neurons, and basically determines how *interesting* in terms of magnitude are each one of the perceived stimuli with respect to each other. Such magnitude will be later used to trigger reflexes and direct the attention of the character.

The interconnectivity and configuration of the attention network regulates the number of features perceptible by a character, and the sensibility to such features (i.e., how *fine* are each one of the input channels). Since perceived information is the basis for the character's world representation, action selection and decision making, the attention network will greatly affect the character's personality. For example: two mice, one with a very accurate hearing sense and other with a fine smelling sense, will present different behavior when the cat is approaching. The first is likely to rotate its head to find the cat's position, while the second might just stand on its back legs and smell the air to verify the cat's proximity. If this last mouse is blind, we can picture a comic scene where the treacherous cat is immediately behind both rodents, the first probably paralyzed with fear, while the second innocently smells the air in several directions without realizing how close to danger it is. Both mouse characters possess exactly the same cognition and expression layers, but a slight difference in their perception layer (a single attention neuron) is enough to produce a radically different behavior.

5.3 Perceiving Variation

The variation module is based on the *Inhibition of Return*[45] effect observed in human perception, which suggests that recently attended stimuli gets inhibited over time. For example, if a fan in my office suddenly starts to turn, my attention gets called by its sudden change, but once a reasonably short amount of time has passed, I will tend to ignore the fan, even when it keeps on turning. The perception of a stimulus x_j gets inhibited when its variation $\omega_{ij}(t)$ is greater than a certain frequency factor ψ_{ij}. A stimulus changes from one perception frame to the next one if the absolute difference of their magnitudes is less than a threshold value. The output of a variation unit ζ_j sensible to stimulus j is 1 if variation is detected for its corresponding feature and the current variation rate is less than ψ_j, and 0 otherwise. This inhibits interest in strongly repetitive phenomena like an object oscillating with constant frequency, a character that maintains a conversation in the background, or a TV set producing white noise. As a consequence, attention is biased toward more unique events like a light bulb that suddenly goes off.

It is important to note that while the attention network measures *intensity* by comparing input magnitudes against a threshold, the variation module detects *change* regardless of the feature magnitude. An oscillating signal will make both the variation and attention modules lose interest after a while, but if the signal becomes gradually stronger while keeping its frequency, the attention module will recognize the change, while the variation module remains the same. Similarly, if the TV set goes off, the attention module will not pay attention to it, while the variation module will immediately notice the change.

The output of the variation module is a *variation stimuli* composed by the magnitudes of all variation units triggered by the perceived stimuli. This stimuli is considered along with the output of the attention network to suggest an attention target to the high level cognitive systems.

5.4 Automatic Reflexes

Automatic reflex triggering is done by a set of *reflex agents*. A reflex agent $RA_l(s_l, f_l(s_l))$ is an independent process formed by a reflex stimuli s_l (a carefully hand–coded combination of features with attention magnitudes) and the reflex function $f_l(s_l)$ over such stimuli. Reflex agents try to match their reflex stimuli against the perceived stimuli (a heat–pain agent, for instance, monitors output from heat–sensible neurons from different parts of the body). If they are similar, the original magnitudes of the input stimuli become the parameters to the reflex function. The reflex function then reviews those parameters and constructs an adequate response. For example, if heat is detected on the character's hand with magnitude greater than a tolerated threshold, the heat–pain agent immediately access the *expression layer* and causes the character to automatically (and involuntarily) move the hand out of the fire. Reflex agents have direct access to the expression layer, so ANIMUS characters execute reflex reactions *before* the stimuli reaches their *brain*, which provides a faster response and a more straightforward methodology for implementing instinctive behavior. Different characters may have different reactions to the same stimuli according to their artistic conceptual design. A cat, for example, will instinctively orient its body to land in its four legs when falling, the same creature will not be able to keep its eyes open if a direct source of light points to its eyes (reflex functions have priority over the high–level cognition systems). Much of the character's personality and behavioral nature can be coded at this level. Reflex agents are similar to the concept of *releasing mechanisms* as described in Maes[53]

In Figure 5.3, an ANIMUS character perceives bullets fired by the user. When a bullet is close to its body, an evasive reflex agent evaluates speed and proximity to move the body away from the bullet. Reflex agents are enough to create characters with interesting (although primitive) behavior, and will strongly shape their personality by the nature of their reflexes. For example, an agile cat will have a fine equilibrium reflex triggered by the minimum variation in the orientation of its body, rotating it to safely land on its feet every time. A dog, on the contrary, might not be that lucky.

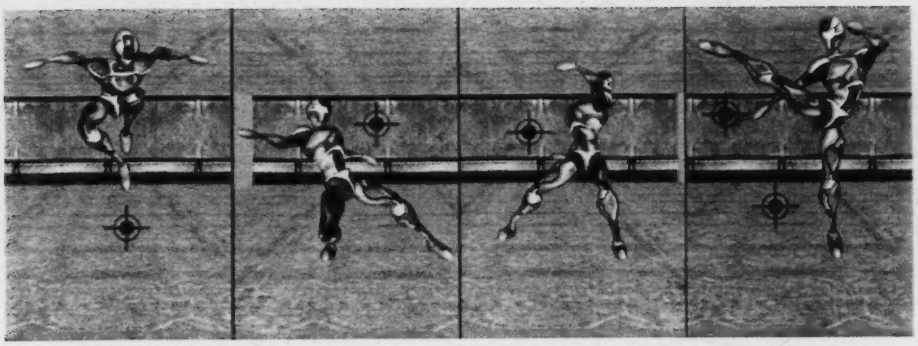


Figure 5.3: An ANIMUS character with a simple reflex agent that moves the body away from bullets

5.5 Conditioned Learning

The conditioned learning memory module is a set of ART[1][11] units. For each reflex agent RA_l that has triggered a response, there exists a unique ART model ART_l that *synthesizes* a sample of the stimuli at the time the reflex was produced. If the character produces a stimuli that results similar to the one that elicited the reflex, a similar reaction with attenuated magnitude will be triggered. To illustrate this with an example consider the classic Pablovian learning case: We would like to train a virtual dog to salivate in response to the sound of a bell. Imagine that the dog contains a reflex agent that produces salivation when the creature is fed. If the user feeds the animal and sounds a bell at the same time, the ART module synthesizes a stimuli that will contain both the sound of the bell and the information about the food. After some reinforcement made by repeating the same stimuli, the user will cause the dog to salivate by just sounding the bell because the bell's stimuli matches, to some extent, the template contained in the ART model of the salivation reflex. In other words, the virtual dog *suspects* they are related and therefore triggers the agent. When the food appears to confirm this relation, the ART module is updated and the bell's stimuli creates a closer match next time it is perceived.

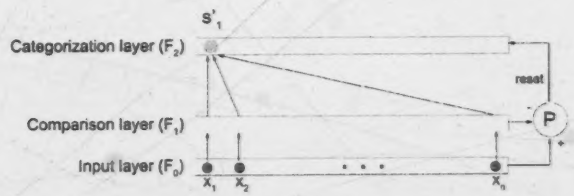


Figure 5.4: ART structure

Figure 5.4 shows the structure of an ART model. The input layer is composed by the stimuli S_i'' coming from the attention network.

[1]Adaptive Resonance Theory.

The functionality of an ART model itself is separated in two different phases:

1. **Synthesis phase**. If a stimuli $s_j \in S_i''$ produced a response from RA_l, a match for such stimuli is searched at F_2 of ART_l. If no match is found, a new neuron containing a synthesized version of s_j is created. If a match is found, the pattern in such neuron is refined.

2. **Comparison phase**. If s_j did not produce a reflex response and variation for this stimuli is strong (i.e., this is a new stimuli in *temporal* terms), we try to find a match in the same way than the previous phase. If no match is found, no action is taken (i.e., we had no previous information about the stimuli); otherwise, the synthesized version of the matching neuron and the input stimuli are combined to create a new stimuli s_j' that is passed to RA_l, in other words, we try to trigger a conditioned or learned response.

The functionality of this module is inspired by the work of Mumford[47] and assumes that events in the virtual world are not arbitrarily produced but respond to hidden variables. Synthesizing coded signals for the creation of comparison points in a feed–forward, feed–backward loop create knowledge about these hidden variables and *teach* the character to react to events in the immersive environment. The ART model was chosen because it is a simple way for creating, storing, and comparing information *templates* with incoming stimuli from the world. It is important to note that conditioned learning by itself is not enough to shape the personality of a character, since we are still dealing with purely *reactive* and linear behavior. This module contributes to the natural and lifelike behavior of the character, but more *ad hoc* high–level models of conceptual learning must be adapted at the cognition layer for influencing proactive behavior and personality.

5.6 Attention Shifting

When the perceived stimuli reach the attention shifting module, we already have enough information coming from the attention network and the variation module. In addition, we have performed reflex triggering and conditioned learning. What is left is to *prioritize* the perceived set $S_i'(t)$ to obtain a new set $S_i''(t)$ that suggests a new attention target to the high–level cognitive systems. This can be done in several ways depending on the nature of our characters. One option is to multiply $S_i'(t)$ by the variation and attention vectors, but this would completely inhibit stimuli discarded by the variation module. What seems to be an effective approach is to combine the variation and attention vectors into a single **attention shift stimuli** $S'_{shift_i}(t)$, and pass both sets to the cognition layer.

Figure. 5.5 shows the *complete* attention shifting procedure, which is in fact executed at both the perception and the cognition layers. The perception layer proposes an *early* attention target to the cognition layer. In animals, for example, attention frequently shifts to this proposed target since they follow an instinctive behavior (it is relatively easy to distract a dog by making

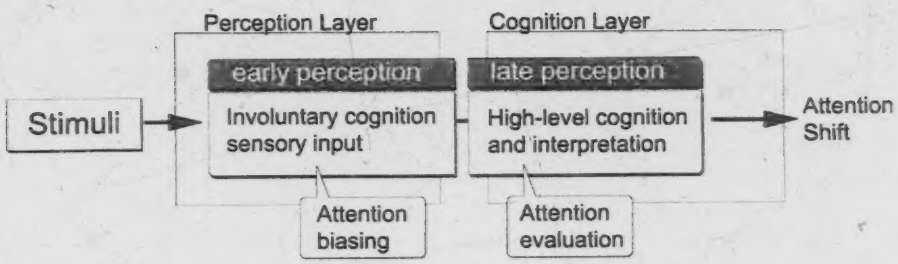


Figure 5.5: ANIMUS Cross–layer attention shifting model.

noise, showing it food and such). In humans, however, it is possible to shift or maintain attention on a specific target for more complex reasons; I might be paying more attention to a news narrator on TV than to the smell of a pizza in the kitchen table. I would risk the assumption that, as the high–level cognitive capabilities and intelligence of an organism become more advanced, its attention shifting depends more and more on high–level reasoning than on early perceptive triggering[2]. Similarly, the more primitive an organism becomes, its attention targets are more likely to be chosen by its instinctive perceptive features alone.

The same applies to our characters: if we are to design an animal–like creature, its attention will be generally shifted toward salient stimuli suggested by the perception layer alone (imagine a puppy that happily runs to whoever calls its name), while a more advanced character (human or otherwise) will show how its perception system bias its attention target, but ultimately will chose the one that proves more important given its internal rational processes, personality or emotional state. For example, if character A is angry with B, he might deliberately chose to ignore her, and even decide to pay attention to events that are obviously not interesting just to elicit anger in her. As the reader can see, personality is again identified by the way a character handles attention shifting and selective reactions to events or other characters in the environment.

5.7 Creating a Perception Layer for Alebrije

Alebrije's perception system has the following characteristics:

1. Alebrije is able to see characters that remain within 180 degrees of his peripheral vision.
2. Alebrije is able to hear characters that are within a radius of 4 meters.
3. Alebrije's ability to change his skin color is an automatic reflex

[2]Some reflex agents might even be overridden by cognitive processing. A character could choose *not to look* something that is creating noise, for example.

4. Other reflexes include the impulse to make visual contact with stimuli that attracts his attention, and move his body to avoid collisions with dynamic objects.

Alebrije's attention network is very simple. It contains one attention neuron that responds to sound, and two attention neurons that are sensitive to position and color, respectively. Features like a character's velocity or relative distance are calculated by Alebrije's cognition layer based on position perceived at different times. ANIMUS provides a special modeling language to declare the perception network[3], which in this simple case contains only one layer and three neurons:

```
<attentionNetwork>
<!-- the decay of the attention network is set to 50, allowing 2.5 seconds for  -->
<!-- attention atenuation if the perception layer samples at 20 times per sec   -->
<layer>
 <neuron ft='SOUND' dc='30' w='1.0' u='0.2' t='0.4' mt='0.02' cw='false' cu='false' out='true'/>
 <neuron ft='COLOR' dc='30' w='1.0' u='0.2' t='0.4' mt='0.2' cw='false' cu='false' out='true'/>
 <neuron ft='POS'  dc='30' w='1.0' u='0.2' t='0.4' mt='0.2' cw='false' cu='false' out='true'/>
</layer>
</attentionNetwork>
```

This declaration constructs a single layer containing three neurons. The first parameter lists the name of the perceived *feature*. The next parameter indicates decay δ in frames, required to cause the neuron to converge to zero; w is the initial input weight $w(1)$; u is the learning rate η; t is the threshold θ, and mt is the maximum threshold for this neuron. cw indicates *constant threshold* if its value is *true*, and the same applies to cw (*constant umbral*); If *outp* is *true*, the output of this neuron is considered for the attention network's output vector. Remember that some neurons might be just intermediate steps for other neurons in the network. If we wanted to create a neuron sensible to a feature that cannot be directly sensed, we can create it from combining output from other neurons. Imagine, for example, that we want to add a feature called *DANGER_SENSE*, similar to the ability of the flies to link visual input directly to their motor system, we can take the information of the sound and position neurons and amplify it to add a signal that will be detected by some *escape* reflex. This would be done in the following way:

```
<attentionNetwork>

   <layer>
      <!-- ...this is equal to the first layer previously declared ... -->
   </layer>

   <!-- now, we add a second layer -->
   <layer>
      <neuron ft='DANGER_SENSE' dc='15' w='1.0' u='0.2' t='0.4' mt='0.02' cw='false' cu='false'
       outp='true'>

            <!-- here we link this neuron to neurons on the previous immediate layer -->
            <arc prev='SOUND' w='2.0'/>
            <arc prev='POS'   w='2.0'/>
      </neuron>
   </layer>
<\attentionNetwork>
```

[3]For a detailed description on the network modeling language definition, please refer to Appendix B.

The second layer contains the single neuron *DANGER_SENSE*, which is linked to the *SOUND* and *POS* neurons of the previous layer with equal weights of 2. We can turn off the *outp* value of the neurons in the first layer, since our fly character would only need to implement a reflex agent for the *DANGER_SENSE* neuron. Remember that every reflex detonated in the perception layer is automatic since it has direct access to the expression layer.

To use the perception layer, Alebrije's implementation will override a function called *OnUpdatePerception*[4] where we have a chance to read the world's blackboard. The following code will read sound and position for all perceivable characters and pass it to the perception network:

```
1   void int CAlebrijeCharacter::onUpdatePerception()
2   {
3           // 1. Create some variables to store information
4           CStimuli* pWeightVector    = new CStimuli();
5           CStimuli* pVariationVector = new CStimuli();
6
7           CDataRecord* pDataRecord = 0;
8
9           // 2. Get a pointer to the virtual world's blackboard
10          BLACKBOARDITERATOR pBlackBoard = m_pVirtualWorld->getBlackboard();
11
12          // 3. For each available character
13          do
14          {
15                  // 3.1 Get the data record from this character
16                  pDataRecord = pBlackBoard->m_pElement;
17
18                  // 3.1 Add this data record to our network
19                  m_pAttentionNetwork->addDataRecord(pDataRecord);
20
21                  // 3.2 read the next data record
22                  pBlackBoard = pBlackBoard->m_pNext();
23
24          }while(pBlackBoard);
25
26          // 4. Activate the attention network
27          m_pAttentionNetwork->weightStimuli();
28
29          // 5. Now we can read the output vector for each of the
30          //    perceived characters, for the attention network and variation modules
31          CFeature speed(SPEED,0.0f);
32          CFeature position(POS,Vector(0.0f,0.0f,0.0f));
33          while(m_pAttentionNetwork->getNextWeightVector(pWeightVector,pVariationVector))
34          {
35                  // read the speed intensity
36                  pWeightVector->GetFeature(speed);
37
38                  // ... Here goes the code that processes the speed intensity ... //
39
40                  // read the variation for speed
41                  pVariationVector->GetFeature(speed);
42
43                  // ... Here goes the code that processes speed variation ... //
44
45          }
46  }
```

Lines 4 and 5 declare *stimuli* objects, which are feature containers that we will use to read the output from the perception layer's systems. Line 7 creates a

[4]See Chapter 8 and Appendix C.

pointer to a *data record* structure, which is the basic unit of the information contained in the world blackboard. Line 10 queries the virtual world for a reference to the blackboard itself, which is used to recursively read data records, that is, the information expressed by other characters in the immersive environment. Line 19 recursively adds all perceived creatures to the attention network, which is the only preparatory step that must be taken to feed the perception layer. After all the characters have been provided to the network, the perception layer itself is activated in line 27. This step puts to work both the attention network and the variation modules. All that is left is to review the result, which can be iteratively read from the attention network object as seen in line 33. The *getNextWeightVector* function returns both the resulting stimuli from the perception network and the variation modules. We must call this function once for each stimuli that has been fed into the perception layer. In line 36 we read the intensity assigned to the *speed* stimulus or feature in the stimuli. The same can be done for the variation stimuli. Usually, we pass the resulting stimuli to one or more *reflex agents* in a code that would look like this:

```
1
2   // get the attention and variation stimuli for a given character
3   m_pAttentionNetwork->getNextWeightVector(attention,variation)
4
5   // Trigger the reflex agents. Remember that reflex agents are functions
6   // that read stimuli
7
8   myReflexAgent1(attention);
9   myReflexAgent2(attention);
10  myReflexAgent3(attention);
11
12  // optionally, we might want to combine both vectors and pass the
13  // result
14  CStimuli* resultingStimuli = prioritize(attention,variation);
15
16  painReflex(resultingStimuli);
17
```

Reflex agents are individual functions (or even objects) that take stimuli as input. They have direct access to the expression layer, and react in unique ways to the results from the perception layer. As an example, consider the agent that reacts to sound produced by some character and turns to face it if its location can be accessed:

```
1   void ReactToSoundReflex( CStimuli* attentionStimuli )
2   {
3           // detect a feature named SOUND in the stimuli chain since we only
4           // react to that stimulus. Alebrije can only hear characters
5           // closer than 4 meters, therefore sound stimulus from a distant character would
6           // not be contained in the perceived stimuli.
7           CFeature sound(SOUND,0.0f);
8
9           if(attentionStimuli->Get(sound))
10          {
11                  // we are able to hear this character. How much attention?
12                  if(sound.Value() >= SOUND_THRESHOLD)
13                  {
14                          gazeAtCharacter(getCurrentCharacter());
15                  }
16          }
17  }
```

Line 5 creates a feature linked to the *sound* property, which is used to query the perceived stimuli. If we have indeed perceived the sound of a character (remember that expected features are not necesarily produced by existing characters) we analyze its attention level, if it is higher than a certain threshold we trigger a reflex that will make the character turn and see the sound source. Functions that can be used to animate Alebrije in such a fashion will be discussed later in this chapter. In this example we are assuming that *getCurrentCharacter* will return a pointer to the perceived character that originated this stimuli. In addition to this, if Alebrije hears the character but cannot find its position (because the POS feature is not accessible) he could show himself confused, stand up and try to look for the sound source.

To make visual contact with characters that attract his attention, Alebrije contains a specialized *gaze reflex*. This reflex can be triggered both by the perception and cognition layers. When a character gets too close, Alebrije turns his gaze upon it automatically, otherwise, deciding which other character will receive Alebrije's attention becomes a cognitive process. Other reflexes, like changing its skin color, can be triggered given the color feature of the ground he is walking. The code for animating Alebrije's body and a more detailed implementation of this particular reflex will be discussed in Chapter 7.

5.8 Design Tools

Generating XML–based definition files for attention networks can be a relatively easy task for simple cases like Alebrije, but it can get tedious and complex as we require more complex perception systems. For this purpose we have created a graphical application that helps designing these elements in an intuitive environment, and outputs definition files that can be loaded directly into ANIMUS, and fine–tuned by the designers.

Figure 5.6 illustrates the simulator that allows to graphically place layers and attention neurons to create the attention network. Links between neurons can be added, deleted, or modified with mouse–drag operations, and each neuron can be individually configured or adjusted by changing its parameters. The application allows to save and load networks in the ANIMUS XML–derived format.

Figure 5.7 illustrate how one can manually create stimuli to test the behavior of the attention network in *realtime*. It is possible to define different concurrent characters, each one producing as many stimulus as desired. The user can easily *draw* the magnitude of each feature across a number of perception frames to simulate activity in relation to time. Stimuli *sets* can be loaded or saved for future reference or testing. After a stimuli has been designed, it can be put through a network created in the interface shown in Figure 5.6 to evaluate the reaction of the system.

5.9 A Simple Perception Simulation

To illustrate the functionallity of Alebrije's perception system, we have created a simple simulation that includes Alebrije and a number of *Noise*

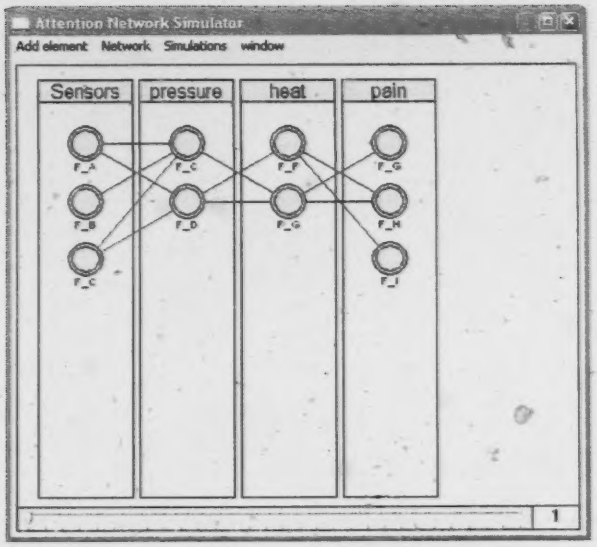


Figure 5.6: Attention network graphic design tool

Insects, whose only function is to emit noise as they are moved by the user or the computer. Alebrije's behavior is based on the functionality already described in Section 4.3.

Once the scenario is set, the user can create any number of noise insects and move them in the virtual world. Alebrije is free to react to those who best stimulate his attention. For this experiment we use 2 noise insects: one is put to oscillate in a cyclic fashion by the computer, while the other is controlled by the user. The perception rate is set to 20 samples per second, Variation threshold is 4 samples/sec, and the decay for the noise-sensible neuron is set to 30 samples. Figure 5.8 shows the noise magnitude along 350 frames as it is created by both insects. The dotted line belongs to the automated noise insect and the continuous line is the user-controlled insect. We will refer to the computer insect as $C1$, and the human controlled insect as $C2$.

When $C1$ begins to move, a stimuli with its noise magnitude is perceived by Alebrije's Attention Network. The noise neuron is excited with an output of aprox. 1 since its initial threshold is 0. The Variation Module detects the change and also outputs 1. The reflex stimuli of the "stare" reflex agent is set to trigger when $noise_t > 0.7$, so alebrije wakes up and stares at $C1$. At this point, w_{noise} begins to decay and θ_{noise} becomes equal to the perceived noise. As $C1$ keeps changing more than 4 times per second for the rest of the simulation, the variation module output quickly becomes 0 for this creature and never finds occasion to change. The cyclic peaks of noise produced by $C1$ take Alebrije's noise neuron to produce the output we observe in Figure 5.9 until the weight decay causes the neuron to *adapt* to this pattern and its output converges to 0 after frame 100. At this point Alebrije looses interest and goes back to sleep. $C2$ joins the simulation at frame 135 with a noise magnitude two times greater than θ_{noise} (which had adapted to the peak noise

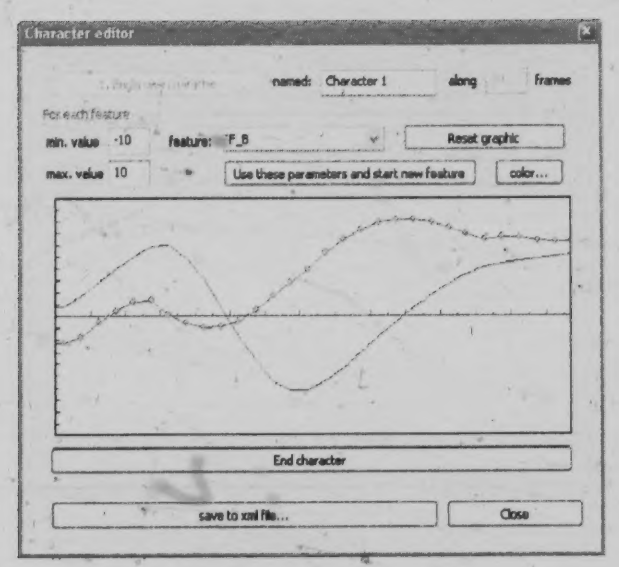


Figure 5.7: Graphic stimuli editor

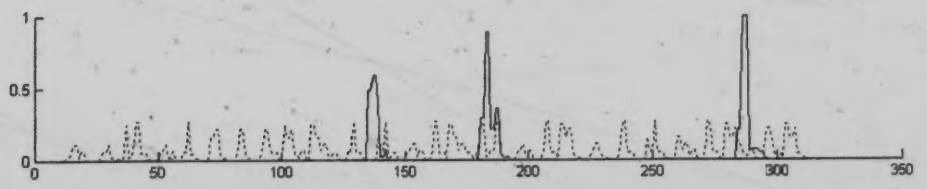


Figure 5.8: The unmodified noise stimuli created by the insects.

of $C1$). The noise neuron reacts and outputs 1, adjusting its w and θ values. The Variation Module also recognizes the change and outputs 1 for $c2$, but remains in 0 for $C1$. Alebrije wakes up and turns to face $C2$, who immediately stops moving. Alebrije then turns to $C1$ since its noise magnitude is higher in comparison to $C2$ (which is now static) and also was the last interesting bug before $C2$, but around frame 170 the noise neuron adapts again and Alebrije goes to sleep. This same event repeats two more times at frame 178 and 284. Since $C2$ has a lower variation rate, the variation module outputs 1 every time it moves. If at frame 284 (when $C2$ moves for the last time) $C1$ and $C2$ had produced the same magnitude, the variation stimuli would have broken the tie favoring $C2$, taking Alebrije to face it instead of $C1$.

5.10 Discussion

The current implementation of the perception layer succeeds to comply with all its initial requirements and objectives. Because of its flexibility, this layer can be considered as a generic solution for many different flavors of characters and environments (this, unfortunately, cannot be said for the cognition layer).

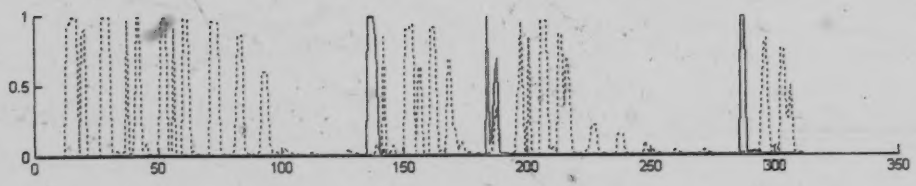


Figure 5.9: Reaction to both stimuli by Alebrije's Attention Network

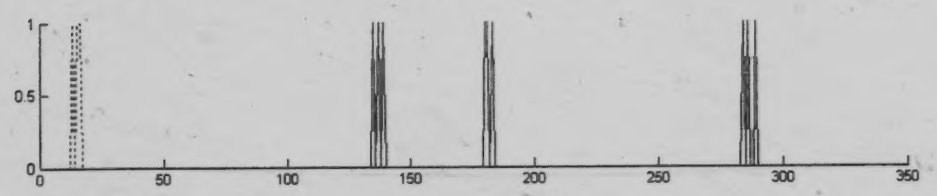


Figure 5.10: Reaction to both stimli by Alebrije's Variation Module

It is possible to adapt the perception channels of ANIMUS characters to react to many kinds of stimuli in the environment, produced by other characters or by human audience through different types of sensors. The perception layer allows the designers to create *ad hoc* perception and instinctive reaction mechanisms for their creatures, providing an excellent starting point for an advanced believable synthetic character. Animal–like characters will have carefully designed perception layers and very simple cognition layers, while more complex characters require the design of effective high–level cognition systems, and will not abandon their behavior to a simple set of rules.

The perception layer alone is able to provide characters with primitive animal–like reactive behavior, similar to those of other projects that claim to implement *complex* AI algorithms. I am not claiming that this implementation is better, but demonstrating that a very simple system is enough for generating reactive behavior. The real challenge resides on the high–level cognitive features that will bring personality, goals, emotions, social behavior, *et cetera*.

Some important aspects to note about the perception layer are:

- None of the values of $S'_i(t)$ are altered in the perception process. The reflex agents match their reflex stimuli against the result from the attention network, but work with the original perceived magnitudes, and the cognition layer receives those same magnitudes along with the newly created $S'_{shift_i}(t)$ set. Magnitudes resulting from the attention network and variation module are merely for prioritization of information.
- Reflex agents can be considered at different levels. Low–level reflexes cannot be overridden by the cognition layer, examples are reactions to pain, light, and heat. High–level reflexes might be inhibited under certain conditions by high–level rational processes. For example, a cat that is hunting a mouse can avoid getting distracted by the smell of food or other minor stimuli in the environment.

Figure 5.11: The Alebrije character while sleeping (left), looking at computer's insect (center) and turning to face the user's insect(right)

- We have previously discussed the difficulty of learning in an open environment. The ART memory module is still experimental.

A current disadvantage is that there are many configuration variables that must be manually tuned for obtaining optimal results with different kinds of characters. We are considering the possibility of creating a set of templates that would save developing time and provide a starting point for designing known types of characters.

Chapter 6

Cognition

Voici mon secret. Il est très simple: on ne voit bien qu'avec le coeur. L'essentiel est invisible pour les yeux.

-Antoine de Saint-Exupéry, "Le Petit Prince"

In the ANIMUS architecture, the cognition layer is the module that truly makes characters *unique* by giving them advanced functionality that goes far beyond simple reactive or rule–based behavior. Cognitive tasks include high–level features like personality, emotions, plans, roles, intention, goals, decision making, social and individual notions, temporal and long–term memory, and basically every aspect that denotes an autonomous and self–sustained character. The cognition layer is, to describe it in a romantic way, the *spirit* inside the character, the spark that makes it behave not as a robot or a reactive creature, but as a self–conscious, independent, *lifelike* being.

This challenge is, of course, of enormous proportions. If we claimed we had a definite method to implement all of the previous features we would be surely lying. *There is no unique solution in this research area*, and probably there are not even completely satisfactory attempts. As we mentioned before, different characters require different cognitive features in terms of nature and complexity. It is hard to find a *one–size–fits–all* cognition model. For this reason our proposed cognitive layer is:

1. A *place* to implement high–level features in synthetic characters, whatever their nature, requirements or methodologies be (A place to implement the *brain* of the character).
2. A high–level abstraction layer to interface with the character's perception and expression layers (a connection to the character's *body*)
3. A set of *tools* to help designers perform further research, or implement some of the mentioned features (A set of classes, objects and methodologies to facilitate research and implementation).

This cognition layer provides a place and a base for testing many different models. Independently of the cognitive model we wish to implement, the perception layer will supply all the necessary information, while the expression layer will be also there to help convey to the user the internal and external high–level processes of our characters. We might go as low as implementing a rule–based approach for the most simple animal–like creatures, or consider more advanced models like the EM architecture proposed by Reilly[57]; complex plan–based architectures like the one described in Gratch[23]; an adaptation for real life robots like Velázquez suggests[71]; Strategy representation as in Gordon[19, 20, 21]; and in general any kind of model that best adapts to the particular needs of our characters.

In other words, the perception and expression layers provide a propitious body, while the cognition layer favors the creation of the character's mind.

6.1 The Role of High–Level Cognition Systems

In human beings, the mind touches and orchestrates every fragment of information perceived by our senses or created by the mind itself. Many of the brain functions are in charge to present a reality, and have the ability to interact with it, to be aware of ourselves and the rest of the world. It is yet unknown how all of these functions are performed, but it has been proved that our personality, our emotional and rational self, our interpretation of the world, as well as our notion of self–consciousness depends on the many tasks performed by our brain. In the ANIMUS context, such tasks are *also* called *High–level* cognition processes.

Some of these processes are visible. For example, we can perceive the emotional state of other people, we can ask and understand their plans and motivation, it is possible to know their personality, their opinions. Other processes remain hidden, like someone's *real* motivations for a particular action, the opinion others have about ourselves, the many options a friend is considering before making a decision. In any case, it is because of these high–level processes that we are complete rational and emotional beings. But more important to this research is the idea that, because we observe high–level cognitive processes in other people, *they also appear as autonomous, self–conscious living beings.* Perhaps the secret behind the illusion of life in animated or fantastic characters is that they appear to have their own internal and external cognitive processes. They doubt, they fear, they trick others, have plans, make friends, establish alliances or betray, they are pleased by certain events and altered by others. In other words they too, although *unreal*, seem to have a mind of their own, performing similar process to those of our human minds. When relying in expression only, the actions and movement of animated characters help to convey this illusion of life to the audience. Thomas and Johnston[68] mention twelve fundamental principles of animation required to make a character appear as if it had its own thoughts, emotions, intention and personality, and therefore appear *alive.*

What happens with *interactive* characters? The illusion is no longer possible with just animation (Hence many characters in video games and other interactive media fail to appear alive no matter how good their animation is). To create the illusion of life in interactive characters, it is necessary to provide some high–level cognitive functionality. The character must be able to create its own plans, show internal conflict, remember and forget, be tricked, distinguish good from bad, and in general convey to the audience the illusion of a *mind* and a *spirit* working inside its body in real time.

Because a set of rules will only suffice for the most primitive characters (animal–like, deterministic behavior), We provide a set of tools to facilitate the implementation of high–level processes and benefit from the functionality of the other two ANIMUS layers. This chapter describes three tools that should help the designer implement the required processes according to the character's own nature and needs. These tools are the *Driver system*, the *Dictionary of characters* and the *Temporal memory model*. After explaining each one in detail, I will provide some insight and comments about their integration.

6.2 Driver System

Numerous emotional and behavioral architectures are based on the use of drivers, or at least apply them extensively. A driver is basically an accumulator that keeps track of a certain magnitude that represents the state of a particular feature. Drivers can be used alone or within a weighted network so that they inhibit or promote each other. The magnitude of a driver can be controlled by a specialized external function or/and possess its own grow and decay behavior. They can represent emotions like joy, anger or frustration; sensations like hunger, pain or tiredness, sexual arousal, and virtually anything that can be measured and has a value that changes in time. An excellent example of a practical driver–based system is shown by the creature of *The Sims*[6]. In this game, virtual characters have a set of needs that increase as a function of time or other factors, and a wide array of actions the user needs to perform to satisfy drivers with dangerously high magnitudes (The user must feed his hungry characters, or entertain the bored ones). More complex driver systems can be seen in Velázquez[71] and Kline[34].

The ANIMUS framework provides a flexible driver implementation. A driver is defined as $D(\phi, \upsilon, \theta)$, where ϕ is the *homeostasis* state or equilibrium value, υ is an *umbral value* that defines an area around ϕ in which the magnitude of the driver is considered to be normal or desirable, and θ is a threshold value that usually indicates a maximum tolerance point. When drivers are interconnected in a network or group, the magnitude of driver D_i at time t is simply written as:

$$D_i(t) = \sum_n (D_n(t) \times w_{ni}) \tag{6.1}$$

Where $D_n(t)$ is the magnitude of drive D_n at time t, w_{ni} is the weight from drive D_n to drive D_i, and n denotes all drivers connected to D_i.

Message	Driver Magnitude Events
DM_THRESHOLD	Surpasses threshold
DM_SIGNCHANGE	Shifts sign
DM_UPDOWN	Increasing to decreasing
DM_DOWNUP	Decreasing to increasing
DM_UP	Increases
DM_DOWN	Decreases
DM_HOMEOSTASIS	Is within homeostasis value

Table 6.1: Callback messages for ANIMUS drivers

An important aspect about the ANIMUS driver implementation is that they are programmed as a callback messaging system. This architecture makes it straightforward for the cognitive system to react to different events coming from a driver or a driver network, without having to be constantly monitoring their magnitudes. Table 6.1 shows some of the messages that can be set in a generic driver.

Messaging simplifies code writing and scripting because we basically wait to be notified when something interesting happens, instead of having to be constantly searching for a particular set of conditions. This gives the architecture an event–oriented approach. As an example, if a character becomes emotionally unstable whenever its hungry, a hunger driver can be set to automatically modify some emotional state, or add a *look–for–food* function to the character's execution agenda whenever hunger strikes. This driver architecture also brings the possibility of creating *specialized* agents that receive events from certain drivers and control the character's behavior in interesting ways, instead of just letting the magnitude of a driver linearly increase or decay by itself in time. Consider a driver that measures *interest* in a certain activity like hunting a prey. If the character is not making any progress (the prey is hidden and won't come out) an agent might decrease or increase interest as a function of variables in the hunting scenario like hunger level, probability of catching the prey, observed behavior of the prey, presence of other nutrition options, and such; instead of just letting interest decay linearly as long as the prey is hiding until the character loses interest (if our predator is extremely hungry, we might want it to wait for the prey to show up, starve to death in the attempt, cry, or show its frustration, and not just lose interest and go away).

Although the framework still lacks a XML–derived language for the automation of *driver networks*, the same tool introduced in Chapter 5 has functionality for creating graphical layouts of drivers and their interconnections, change their magnitudes and messages, link them together in a network with simple *drag and drop* behavior, and modify their inner values to see how the complete network behaves. This tool is useful to see the network in action and evaluate its performance.

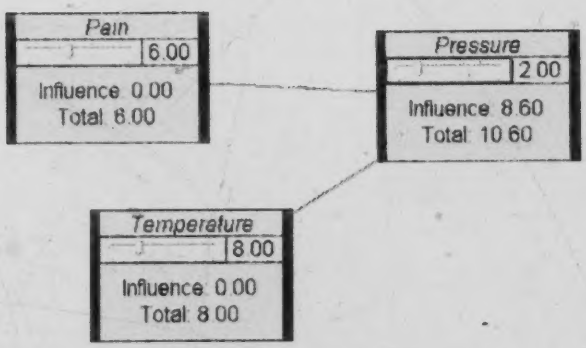


Figure 6.1: Driver network graphic simulation tool

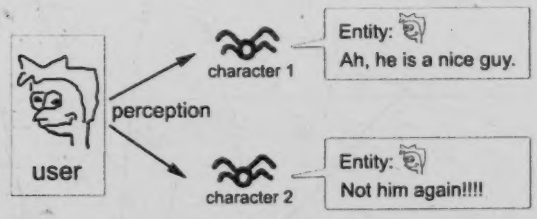


Figure 6.2: Two characters perceiving the same entity.

6.3 Dictionary of Entities and Characters

To favor high–level knowledge representation and modeling, two special data structures are provided. They are *entities* and *characters*. An entity is anything that our creature recognizes as an individual object, living or otherwise, including the human audience. A character is defined as the *particularized* version of an entity from the point of view of the synthetic creature. In other words, an entity is like the picture of a creature, which is always the same for everyone, while a character is the picture plus our own knowledge, beliefs, emotions, annotations and experience about such creature, which might (and hopefully will) change in time. In a simulation with two virtual spiders and the human user, both spiders recognize the user as the *same* entity, but each one could have a different concept of him given their personal experience, and therefore react in a different way to his presence. This simple principle enables to otherwise exact instances of the same object (the spider) to have different knowledge, and therefore different behavior, and create their very own representation and belief structure about the world. In Figure 6.2, two spider–like characters perceive the same entity (the user), but interpret it as a different character.

The dictionary of characters provides ANIMUS creatures with a database that stores particularized knowledge about other creatures and the human user[1].

[1]From the point of view of any ANIMUS character, the user is *also* a character.

In practice, references to entries in the dictionary of characters can be used in other cognitive processes to represent creatures, as demonstrated in Section 6.4.

6.4 A Temporal Memory Model

The temporal memory model is similar to the work of Mumford[47] and Damasio[15] and is mainly based on the theory that knowledge can be acquired, represented, and maintained by coding temporal signals of events that happen in the world. A basic representation of the memory module elements is illustrated in Figure 6.3.

In Section 5.5, we mentioned that events in the environment might not be arbitrarily produced, but due to specific hidden variables and relations between events. To find such information and represent the world state we propose a model that allows to *learn from experience*, detecting patterns as a function of both time and stimuli data. The model itself is very simple: for each known character[2], we synthesize perceived stimuli in a progressive timeline structure, adding an entry each time the world is sensed. We can now analyze two kinds of information:

1. The set of stimuli belonging to all perceived characters at some point in time, which is called a *time frame*, and also represents the state of the world at that time.
2. The behavior history for specific characters within a certain time lapse, called a *time sequence*.

Specialized pattern analysis agents can read the temporal memory both *vertically* and *horizontally* to respond questions like: What characters were present when I entered the room? Given observed events, what is the most probable thing to happen if I perform a certain action? What does character X do every time Y sings? What are the common characteristics of the world and other characters every time the user seems to be angry? What is the average speed of character Y when it is walking over mud? What is the set of conditions that seem to be required for some phenomena to happen? High–level cognition systems can take advantage of information stored in temporal memory objects to create or discover knowledge, and learn about the world and its characters. This model also allows the notion of learning through *episodic memory*, as described by Isla and Blumberg in [28]

Let us try an example: some character believes the user is sad and asks itself what to do to cheer her up. Searching through its temporal memory, it recalls 6 different occasions in which the user was happy. Comparing those 6 time frames, it discovers that a common factor was the presence of a flower. Because there is no flower in the scene now, it goes to fetch one. As the user is still sad, it looks for more information: the position of the flower in those 6 episodes was always the same: by the window, so the character grabs the flower and moves it to the correct location. The user smiles, she likes the

[2]Here, we refer to the concept of character discussed in Section 6.3.

way sunshine reflects on the flower's leaves. If this had not succeed, other analysis techniques could have being applied, like reviewing the character's own behavior sequence and that of other characters prior to the time frames where the user was happy: perhaps the user is sad because some character is not performing in the same way.

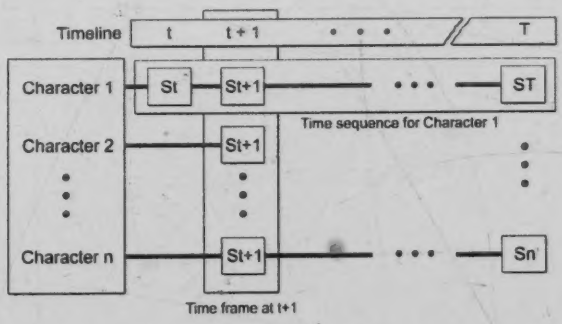


Figure 6.3: The ANIMUS Temporal Memory model.

Stimuli for a given character might or might not be present at a given time, and the system's memory constraints will determine how many time frames to keep. Multiple temporal memory objects can be created, updated and consulted for organizing information and given physical memory constraints an active window may keep only the last n time frames and store information that proves to be relevant in other less frequently updated temporal memory modules. Temporal memory modules are powerful structures that also allow interesting operations like interpolation and extrapolation of stimuli. Some character might try to predict the behavior of another character by extrapolating past similar actions, or use interpolation to deduct what happened between two points in time when its behavior was unknown.

6.5 Creating a Cognition Layer for Alebrije

The cognitive operations performed by Alebrije are the following:

1. Maintain a temporal memory to remember the last immediate state and actions of characters in the scene, in order to have a short-term world representation;
2. Remember every known character (but not their states);
3. Select an attention target if the gaze reflex did not select one automatically;
4. Decide when the scene has become too boring to remain awake;

To maintain a short term (or temporal) memory record of characters in the scene, we use an instance of the *temporal memory object* described in Section 6.4. A temporal memory object is composed by a set of *layers* that keep

track of stimuli belonging to different characters in the virtual world, within a certain time frame (one layer for each character). Remember that entities and characters are two different abstractions. Figure 6.4 shows the dictionary of entities and characters, and a layer from a temporal memory module. The character n in temporal memory is actually a reference to the character in the dictionary, which is composed by the perceived entity and the knowledge and beliefs about that entity. Before we can remember anything about a character, it is necessary to add such character to the dictionary.

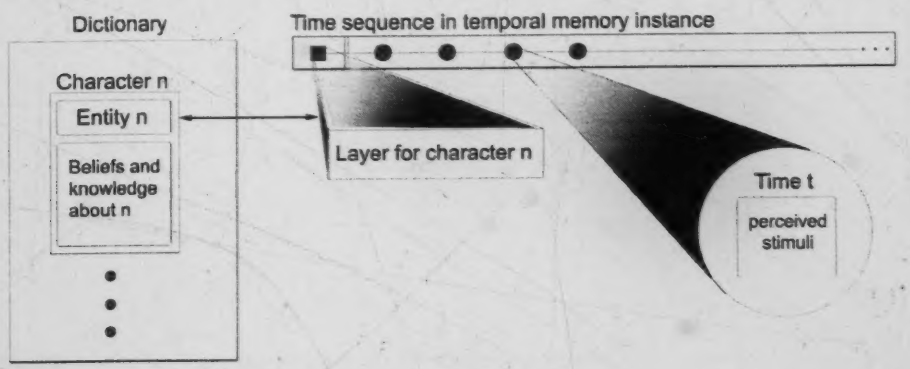


Figure 6.4: Data referencing between temporal memory and dictionary instances.

The following code creates a new character based on the entity contained inside a data record, adds it to the dictionary, and adds a layer in a temporal memory object so that we can begin to *remember* information about the character.

```
1   // We probably know this entity already. Otherwise add a new layer for it.
2   if( !m_pShortTermMemory->getLayer( someDataRecord->getEntity() )
3   {
4           // Create a new character with this entity
5           CCharacter* pNewCharacter = new CCharacter( someDataRecord->getEntity() );
6
7           // Add this character to the dictionary
8           m_pDictionary->addCharacter( pNewCharacter);
9
10          // Add a new layer to some memory model instance, based on our newly added character
11          m_pShortTermMemory->addLayer( pNewCharacter );
12  }
```

To add information about a character in the temporal memory, we simply provide a pointer to the character's layer and the stimuli itself.

```
1   // If we don't have a pointer to the layer, we can query the
2   // memory object by providing a pointer to the character or its corresponding entity
3   CMemoryLayer* pThisLayer = m_pShortTermMemory->getLayer(aDataRecord->getEntity());
4
5   // Add an entry at the end of the memory timeline for this character
6   m_pShortTermMemory->addKeyframe( pThisLayer, pSomeStimuli );
```

The temporal memory object maintains information within a certain time window only. Several memory modules with different windows sizes can be

utilized according to the type of information they will keep. For example, one can use a short term memory window for real–time perception, and a longer memory window for storing interesting information about a set of characters. Alebrije's time window keeps a history of perceived information for the last two seconds only, since we do not need to perform any complicated data analysis. At each perception step, Alebrije records other character's position, color, and sound. If a character changes its color, Alebrije is able to notice and trigger his attention agent, but he will not be able to remember which color that same character had three seconds ago. The dictionary, on the other hand, will always remember every perceived character, even if they disappear from the scene

Selecting an attention target can be done at two different moments. The first one is at the perception level by means of a reflex agent, when some character performs an action that attracts alebrije's instinctive attention, like showing high intensity or variation in their expressed features. The second moment is at the cognition level when nothing relevant seems to be happening. In this case, Alebrije must decide which character to pay attention to, or probably just go back to sleep. Currently, the first rule is to check the last character that caught Alebrije's interest, which can be read from the temporal memory object (It is also possible to make the temporal memory window bigger, and try to look for unnatural behavior in known characters). If the last character is not doing anything interesting, Alebrije checks how *bored* he is, and eventually may go back to sleep. To decide if the environment has become too boring, Alebrije has an *interest driver* that increases or decreases as a function of the activity in the environment. When the driver fails to satisfy a certain threshold, it uses the ANIMUS messaging system to let Alebrije know that it might be more fruitful to lay back and rest.

6.6 Summary and Discussion

Implementing high–level cognitive features is certainly not easy, but we must also remember that some characters will require simpler systems than others. An animal, for example, can be just fine with a set of rules, while a clown or the evil genius of a story will have to have more specific abilities. Instead of limiting the cognitive layer to a particular model of emotions and personality, the ANIMUS framework provides the set of tools we have discussed in this chapter, so that the designer has freedom and flexibility to adapt the best set of cognitive elements for each particular case.

Future research will investigate this particular topic. We need to design different models of high–level cognitive architectures for specific cases, emotions, and behavior. Perhaps a scripting language would be useful in this case. As a testbed domain for future research, one should consider the flexibility of a videogame environment. A current disadvantage is that the team of people working directly with the cognition layer needs to have good knowledge of C++ programming, artificial intelligence, and pattern recognition, to effectively implement required algorithms, although the use of graphical tools can be further expanded, and high–end applications for the expression layer are already available (3D modeling tools).

Chapter 7

Expression

"If you have no voice, scream;
if you have no legs, run;
if you have no hope, invent"

-Cirque Du Soleil, Alegria

"There is a special ingredient in our type of animation that produces drawings that appear to think and make decisions and act of their own volition; it is what creates the illusion of life".

-Thomas and Johnston, "The Illusion of Life"

The greatest test for an animated character is to appear in front of its audience and perform in such way that it seems to be alive. This is achieved when its personality, intention, thoughts, and emotions are effectively conveyed by its animation. The public suspends their disbelief and imagine them alive. What we are really believing is that such characters have a mind of their own, in other words, they *appear* to have high–level processes as any other living creature. When we have decided how our character works internally (that is, how its perception and cognition layers will function) it is time for the only chance it will have to truly appear alive: stand before the audience and act!

Creating this illusion is a difficult and challenging art. Not every animated character seems alive, it is required that both the internal cognitive systems and the animation aim to the same purpose: the character must have a strong and robust personality; it must impress the public with its body language (like the way it gazes to other characters, the style when it stands and walks, its tone of voice). Everything must be in concordance to *who* the character is as a whole, or it will fail to be believable.

The expression layer is responsible for showing whatever process is taking part inside the character. If this layer fails, the character will fail regardless

of the quality of any other of its components. We might have an immaculate perception and cognition architecture, probably close to the miracle of a truly artificial form of consciousness (hypothetically speaking), but even then our character will be hopelessly doomed without an adequate expression system that shows its quality to the audience. In the other hand, a very simple rule–based behavior with an excellent expression system will actually make the character interesting to the user at least for a while. A good animated character succeeds at the questions proposed by Ommanney[48] for real actors: Are the characters interesting, lifelike, and vivid? Do you become emotionally involved with them? Do the gestures and movements seem sincere, convincing, clear and properly motivated? Does all of the action help to delineate the characters and their situation for you?

This chapter describes the architecture and functionality of the expression layer. We begin with a quick overview of common 3D animation techniques. Next we will outline the requirements of a basic animation engine, and the theory and inspiration sources behind the implementation. We will comment on the responsibility of both the artist and the designer to create a robust animation system, and will finish with some general comments and discussion.

7.1 Common 3D Animation Systems

Implementations of 3D animation systems come in many sizes, shapes, and flavors depending on their nature and application. Although real life animation systems usually combine more than one technique to achieve their particular goals, the most common approaches are the following:

- **Physics–based and realistic animation controllers**
  To maintain the realism of an object or character, it is common to use real physics and mathematics to generate animation keys in real time. This approach is particularly useful for simulators and other training systems that must resemble the conditions of our world, as well as characters or creatures bounded to realistic constraints. Concepts in this area include *inverse* and *forward* kinematics, laws of force and movement, momentum, gravity, general dynamics, among others. Advantages are that animation is guaranteed to be *realistic* (although not necessarily interesting), there is no need to create animation sequences *a priori* or perform complex motion–capture, consistency and robustness is easy to achieve, and the animation is easy to predict and control. On the other hand, it requires a heavy amount of mathematical operations at each frame, visually it can be quite *boring* since mathematics can hardly encode the artistic requirements of a character, it ends up looking robot–like and artificial, and will tend to be the same for every object or character in the scene (Remember that we are looking for very unique ways of motion for each character). Parisy[51] and Terzopoulos [66] are good examples of these kind of systems.

- **Keyframed animation and Interpolated motion**
  A more interesting kind of animation will come from the interpolation of several kefyrames constructed *a priori*. This approach is widely used in videogames and virtual reality simulations involving artificial characters with artistic goals. Instead of leaving the motion to an inverse

kinematic system or a physics model, a number of keyframes are created by character animators or captured from a live actor to carefully handle artistic constraints and definitions, and make sure the character's movement is adequate to its unique style and personality. The motor system concatenates or selects different sequences of keyframes and performs an interpolated animation between them to make the character react to the environment. The major disadvantage is that such sequences are unmodifiable and play in exactly the same way every time. After a short while they become repetitive, which is drastically damaging for the character's believability. Examples include Cavazza[41], and most 3D videogames available today.

- **Verbs and adverbs systems**
  An interesting variation of keyframed and motion interpolation systems are verbs and adverbs systems. A verb is basically an animation that is constructed offline from sets of examples, hand–coded keyframes, motion capture, or any other available technique. An authoring system allows the creation of transitions between such verbs creating a graph. An adverb is the set of weights or parameters that affect the transition or animation between verbs, providing a higher level of flexibility and control. Interpolation between verbs is usually done by radial basis functions and low order polynomials[12]. Limitations include the fact that all related verbs must be similar (every version of a *hand shake* animation must begin with the same hand and have the same phase and number of steps, for example); and no spontaneous animation can interrupt an active transition (when running, the character cannot suddenly decide to wave its hand or shake its head). These systems provide a better degree of control and blending between animations, but the resulting sequences are still deterministic and repetitive. Examples of systems that modify existing animation sequences can be found in Williams[76] and Amaya[33].

- **Blend–based and Pose–based animation systems**
  These systems are similar to the *in–betweening* technique used by traditional animators: Two carefully designed reference poses are drawn by the artist to illustrate the character in a certain action and situation. Intermediate poses are then added to create the complete animation. The flow of motion is not linear, but follows a path that highlights the character's unique body language. This way, the artist has complete control over the expression of its character. Pose–based motor systems contain a set of artistically designed poses. At a given time, two or more poses are chosen and blended together to dynamically generate the animation. A set of parameters in the pose selection and blending algorithms, and the ability to generate *new* reference poses in realtime, give the programmer flexibility to generate animations that adapt to specific situations, while preserving the character's personality and style imprinted in the reference poses. Blending between one animation sequence and the next one can be done by means of radial basis functions as described by Rose[12]. An excellent example of this type of systems is found in Downie[18].

7.2 The ANIMUS Animation Engine

Physics–based approaches and realistic animation controllers are not useful for ANIMUS because they completely fail to encode the artistic constraints of the characters, while adding overhead due to the amount and complexity of calculations for each frame. Few characters designed with artistic or entertainment purposes in mind use this kind of controllers because, in addition to the mentioned reasons, they are generally boring for an audience that is used to the complex physical interactions of a real world, and therefore have very high expectations. Keyframed animation and interpolation captures some of the artistic elements of the character, but the price to pay in terms of flexibility is too hight, and maintaining believability with such a deterministic system is extremely difficult. Verbs and adverbs systems are a closer match to the ANIMUS standards since they add a semantic layer to the animation engine, providing the possibility of selecting more specific animation configurations and sequences according to information in the world, and the character's internal state. Nevertheless, once an animation is selected its execution is almost as rigid as conventional keyframed systems.

The ANIMUS framework implements a *pose–based* animation engine similar to the works of Downie[18] and Rose[12, 61]. It is composed by the creature's geometry (which can be constructed using any professional 3D modeling package like *Blender*[67] or *Maya*[74]), an internal skeleton structure, and an organized and parametrized set of bone "snapshots" or *poses*. A skeleton is a hierarchical set of transformations (bones) that affect the creature's geometry. A single vertex v can be proportionally affected by more than one bone in the following form:

$$v'(t) = v \prod_i b_i(t) I(b_i, v) \tag{7.1}$$

Where $b_i(t)$ is the ith bone affecting vertex v with a particular transformation at time t, and $I(b_i, v)$ is the influence degree of such transformation over v such that $0 \leq I(b_i, v) \leq 1$ and $\sum_i I(b_i, v) = 1$. The mentioned packages have useful graphical tools to distribute the weights of the character's skeleton in a way that the surface moves like an elastic skin, which greatly adds to the aesthetical appearance of the animations. Once the geometry, internal bone structure, and poses have been created with the 3D modeler, one can write the information and then read it with ANIMUS to feed the animation engine's modules. The framework currently supports *DirectX*'s *X* format[1], but the framework's modularity makes it easy to implement input objects for any other format.

Basically geometry, bones, and poses, are just raw numbers that must be organized to create a transparent animation interface for other layers. Each pose will be assigned a numeric *weight* and placed inside an uniquely named *node*, which is actually a container that keeps a collection of usually related poses. For example, one can create a node called "walking" that contains five poses used to create walking cycles. Each node will contain at least one weighted pose. To generate an animation sequence, we simply select two poses (initial and final states) by passing the name of their nodes and a certain weight that usually encodes a semantic value. We may select poses from the same or

[1] ©Microsoft

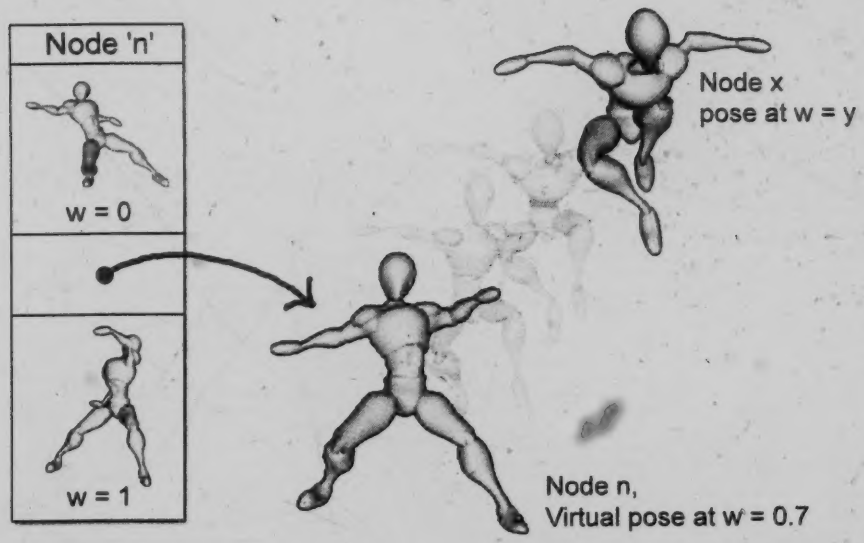


Figure 7.1: Blended animation from Pose $w = y$ at Node x, to a virtual pose $w = 0.7$ at Node n.

different nodes, and when the weight does not correspond to an existing pose, a *virtual pose* is created by realtime interpolation as shown in Figure 7.1. The skeleton is blended across selected poses to animate our character. If an animation needs to be changed or *redirected* to another pose while blending, the current state of the skeleton is considered as the initial pose, and gets blended against the new final pose.

Blending between two different bones can be done using radial basis functions as detailed in Rose[12, 61], bezier interpolation, or simple linear interpolation. When using bezier interpolation, directly interpolating the bones will cause geometry deformation. There are several solutions to this problem. A simple approach is shown in Figure 7.2; given that linear interpolation is guaranteed to maintain proportionality, one can obtain a bezier position using two linear interpolations. This method is not an optimal solution but works for simple scenarios and testing purposes.

Because of pose redirection and automatic virtual pose generation, new animations can be created in realtime, and existing sequences become flexible and may change given the environment's dynamic circumstances. In addition, the main skeleton of the creature can be divided into smaller bone chains (i.e., left arm, right arm, and such) and poses can specify transformations for particular sets of chains. this way, more than one blended animation can be executed concurrently to animate different parts of the body, like walking while waving the hand. A limitation to this approach is that reference poses must have the same bone chains and two concurrent animation threads must not be allowed to blend the same chain.

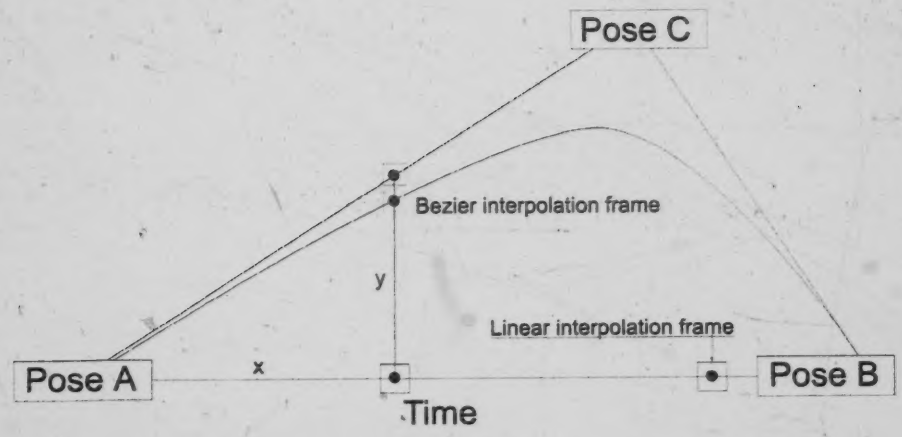


Figure 7.2: Obtaining a bezier interpolation with two linear interpolations.

To instruct the animation layer about available bones, pose nodes, poses, and weights, the framework reads a *physical layer definition file* containing information about the nature and organization of the raw geometric and transformation data obtained from the 3D modeler. This xml–derived language presents two advantages. First, it makes the information more human–readable so that changing, adding, or deleting animation configurations can be easily done, in a portable format that is independent of the geometric information. Second, because of this independence between the raw data and the hight–abstraction animation information, the same definition file can be used *for more than one character*, making it easier to interchange models in a completely transparent way, and generalize animation information. Both characteristics result particularly useful for videogames. An example of this definition file follows:

```
<!-- Minimalistic example of a physical layer definition file -->
<!-- for an ANIMUS character                                    -->
<articulatedCharacter iPose = 'idle'>

   <!-- first, we need to define the bone chains, or skeleton -->
   <chain id='arm' bones='shoulder,elbow,wrist'/>
   <chain id='head' bones='lowerNeck,upperNeck,headBone'/>
   ...

   <!-- a simple head gazing pose -->
   <pose pId = 'gaze'>

      <!-- it contains a node with weight = 1, for facing left -->
      <!-- this pose activates the head, but not the arm, and  -->
      <!-- the bone snapshot to use is number 10               -->
      <node w='1'>
         <skeleton at = '10' set = 'head' />
      </node>

      <!-- the next node sets the head's bones at snapshot 11  -->
      <!-- and sets the head to face right                     -->
      <node w='0'>
         <skeleton at = '11' set = 'head' />
      </node>

   </pose>
```

```
<!-- This pose sets the arm and the head in a 'surprise'   -->
<!-- stance. We can trigger it anytime, but it will          -->
<!-- interrupt any animation we were doing with both the   -->
<!-- head and the arm                                       -->
<pose pId = 'surprise'>

   <node w = '1'>
      <skeleton at = '15' set = 'head' />
      <skeleton at = '21' set = 'arm'  />
   </node>

</pose>

<!-- the following is a gesture that makes the character -->
<!-- look at its right and then look surprised           -->
<gesture gId = 'look_and_be_surprised' fps = '24' >

   <!-- 1. blend from wherever we are, to partially righ  -->
   <!--    side. This will make the head turn              -->
   <walkTo pId = 'gaze' w = '0.2' sec = '0.7' easing = '1' />

   <!-- 2. Now, look surprised                                    -->
   <walkTo pId = 'surprise' sec = '0.3' easing = '0.5' />

</gesture>

</articulatedCharacter>
```

The above code defines two bone segments, one called *head* containing bones for the neck and head, and one named *arm*, with 3 bones. The name of the bones in the definition file must correspond to the name of the bones in the raw data file. There are two poses here, one named *gaze* and another named *surprise*. Gaze has two nodes with weights 0 (right) and 1 (left), so if we animate to *gaze*(0.3) the character will partially turn right. The second pose only has one node, so the framework will ignore weights when blending to it. The *gesture* element is instructing the framework on how to create an animated sequence called *look_and_be_surprised*. We have previously mentioned that animation sequences are usually created in realtime, but they can also be predefined using *gesture* elements. This particular animation will take the character to gaze at its right side in 0.7 seconds, and then stare in amazement in 0.3. The *easing* parameter affects the time interpolation curve as shown in Figure 7.3, this helps creating a more natural motion for certain cases.

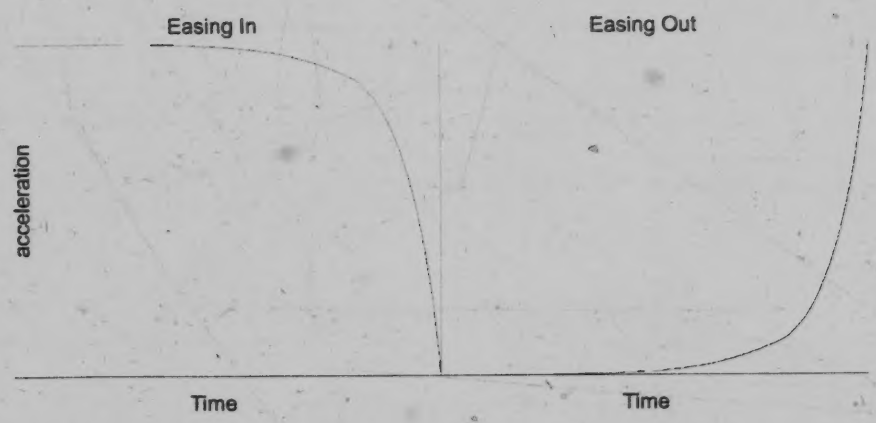


Figure 7.3: Time and acceleration in easing–in and easing–out animations.

Some of the advantages obtained from this system are the following:

- It provides a higher level of flexibility both for the programmer and the artist (as it will be further described in 7.2.1 and 7.2.2), instead of encoding the parameters of an animated action into a rigid sequence, they can be included in the set of reference poses (A *sad–walk* and a *happy–walk* cycle of 5 seconds, each requiring aprox. 120 keyframes, can be substituted by 5 or 6 single poses, and the flexible animation will be generated in realtime).

- A running animation between two reference poses can be modified in realtime: Halfway through the animation between poses a and b we can blend to c without *breaking* the current animation sequence. This allows the creation of *new* animations on demand. In addition, new poses can be also defined in realtime, adding an excellent degree of non-linearity to the system.

- Since the poses are designed with artistic constraints, the resulting animation also complies with the character's conceptual definition. Furthermore, modifying or changing an animation sequence in realtime is also done with reference poses, so it is based on artistic constraints as well.

- It is possible to blend more than one set of poses at the same time, therefore allowing concurrent animation.

- The parameters of the reference poses can be linked to the internal AI methods of the character to produce a wider degree of animations. Instead of selecting between a discreet set of animations like a *happy–walk* or a *sad–walk* alone, the character might want to do something in between. Also, parametrized poses allow the emulation of otherwise costly physic–based methods like inverse kinematics (To grab an object standing at 0.6 meters, the character can blend between a *grab* pose at 0.0 m, and *grab* at 1.0 m).

7.2.1 The Role of the Artist

Before the programmer implements the animation of a character using ANIMUS, it is necessary for an artist or a group of artists to create the geometry and to determine its animation aesthetics. Remember that the character's body language must reflect its personality and emotional state. This is one of the most critical aspects of character design. The following is a proposed set of activities in relative chronological order for the artist or artistic group, after the main conceptual design of the character has been established:

1. Model the character's geometry using a professional or custom 3D modeling package, like Maya[74], 3DStudio Max[17], or Blender[67]. In addition, apply a set of textures and texture mapping to the character. Sometimes the artist is also able to specify a set of special rendering effects linked to the character's materials (particular types of reflective surfaces, bump mapping, illumination, and such), which are later interpreted by the graphic engine in which ANIMUS is built.

2. Create the internal skeleton layout (again, using a high–end 3D modeling package), setting an appropriate weight distribution and the necessary controls for generating the animation.

3. Pose the character and generate a set of pose snapshots. It is also possible to export complete animation sequences. The ANIMUS engine treats the individual keyframes as poses.

4. Export both the character, the skeleton, and the pose snapshots to a file.

5. Write a *physical layer definition file* for the character to organize and parametrize the available poses.

6. Supervise the work of the programmer, so that the character's animation is executed as expected by the artist. The interface for the user can also be designed by the artists using tools like the InTml language[49].

7.2.2 The Role of the Designer and the Programmer

The programmer uses artistic resources created by the artists to put together an interactive character. It is the responsibility of the programmer to customize the three layers of the ANIMUS framework in concordance to the artistic design, so that the character looks, animates and behaves as expected. The programmer must ensure that the perception and cognition layers construct and activate the correct animation sequences, in other words, generate a believable behavior. Usually different teams of designers and programmers work in different aspects of a character. For example, a group of graphic programmers might devote themselves to the creation of some particular graphic engine for the virtual reality environment and its creatures; an AI team might implement the character's internal perceptive and cognitive systems, while another group makes sure the physical interfaces with the user produce the adequate input.

Both artists and programmers must work together and maintain open communication channels. Too many projects have failed precisely because work

done by one or another are not compatible or assume inexistent functionality or methodologies.

7.3 Creating an Expression Layer for Alebrije

Alebrije's expression layer is composed by his 3D model, a set of bones and a group of poses. Figure 7.4 shows his 3D body from different perspectives and poses, and Figure 7.5 illustrates the bone layout.

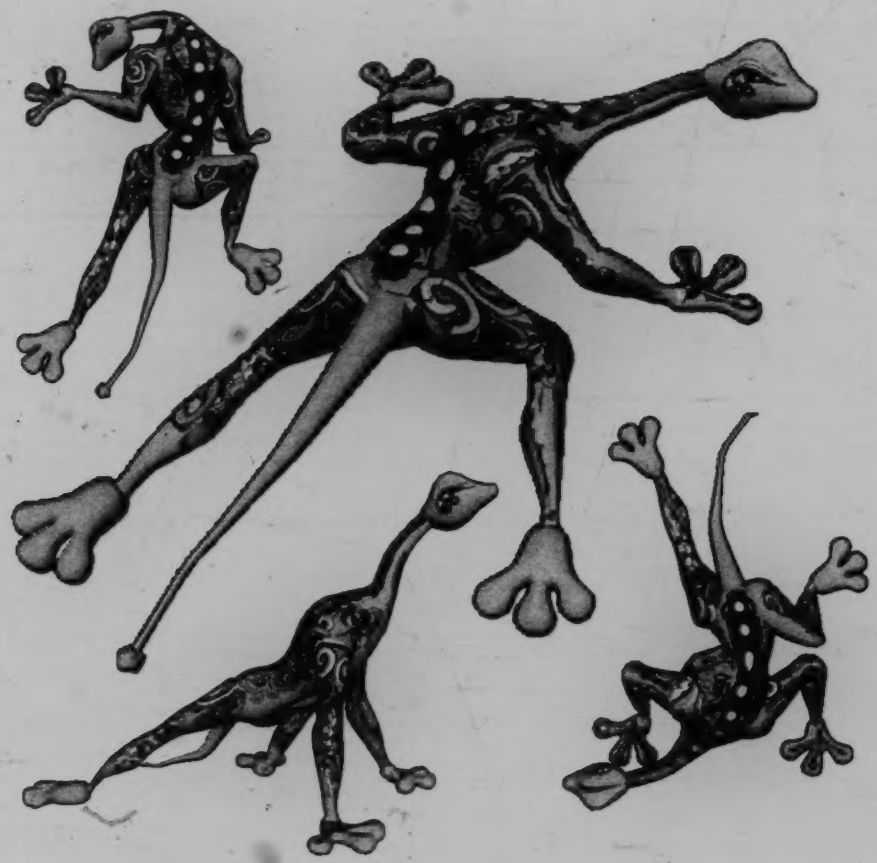


Figure 7.4: Alebrije, from several different perspectives.

Creating Alebrije's expression layer is as simple as indicating the names of the geometry and physical layer definition files. The function used for this task is named *LoadExpressionLayer* and receives parameters from the mentioned filenames[2]. Alebrije contains about 18 different poses, for simplicity let us consider only the following:

[2]Class *CAnimusCharacter* calls this function for us. See Chapter 8.

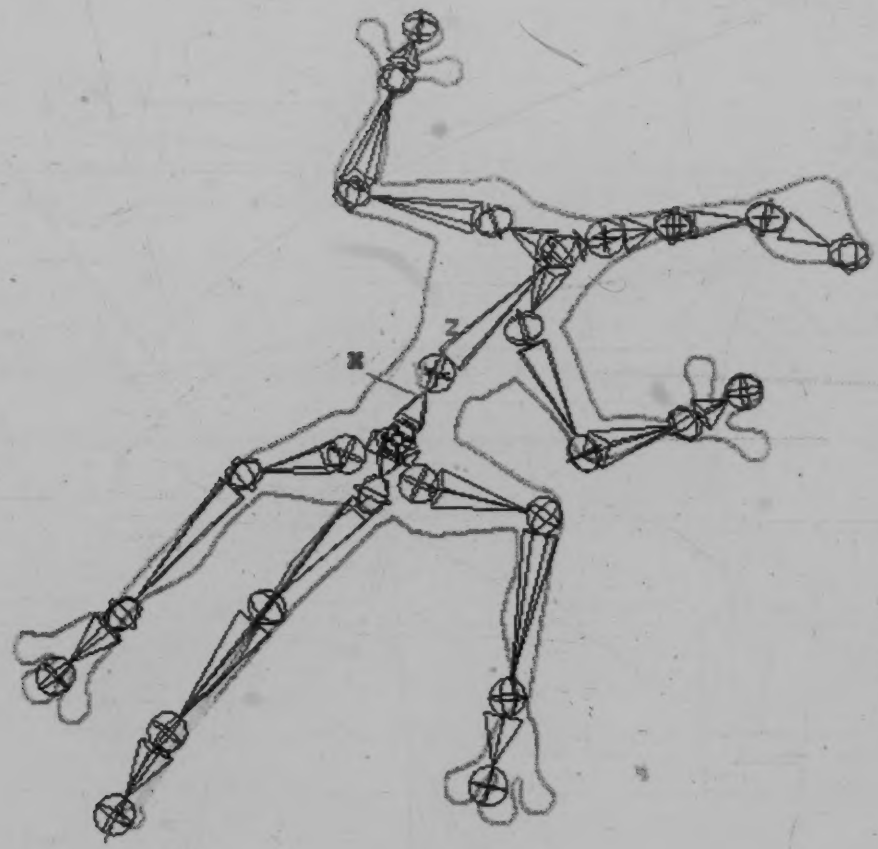


Figure 7.5: Alebrije's bone layout.

- **Alert**: 1 node. Alebrije is awake and gazing forward.
- **Sleep**: 1 node. Alebrije lays down on the floor, sleeping.
- **gaze_long**: 3 nodes. Alebrije looks at objects that are at long distance. Pose weights are 0, 0.5 and 1, for gazing left, center, and right.
- **gaze_short**: 3 nodes. Alebrije looks at close objects. Same weights and descriptions than the previous pose.

Although the animation engine can be queried for pointers to specific poses in realtime, it is always a good idea to store references to the most used ones (if we know them). This can be done as follows:

```
1  // create pointers for alert and sleep poses
2  CPose* pAlert   = getPoseByName("Alert");
3  CPose* pSleep   = getPoseByName("sleep");
```

To illustrate how to trigger an animation, suppose Alebrije has detected a character standing far away. It is desired to animate him in such way that his face turns to gaze at this character. If Alebrije stands at position $x = 0, z = 0$ facing $x = 1$ (Using a left–hand coordinate system), and the other character is located at $x = 5, z = 5$, we must interpolate from our current pose to the *gaze_long* pose with a weight of 0.75. Assume that Alebrije is initially at *sleep* pose. The code to perform the animation is very simple:

```
1   // Animating from sleep to gaze_long with a weight of 0.75
2   // set the animation options:
3   setOptions(AMMSG_ENDSEQUENCE);
4
5   // set the animation sequence
6   setKeyframe(pSleep,0,pGazeLong,0.75f,fps,gfps,0.5f,0.8f,ATHREAD_BLENDED);
```

Line 3 contains a function that sets different states in the animation engine. In this case is instructing it to perform this animation only after other previous animations have ended (as opposed to playing it concurrently). The function *setKeyframe* at line 6 is the one that creates the animation sequence. The first parameter is the initial pose, followed by its weight; the third and fourth parameters are the final pose and weight. The next to parameters are the current frames per second provided by the hardware, and the frames per second at which we need the animation engine to run. The next parameter is the duration in seconds for the animation, then the ease–in/ease–out float, and particular configuration values for this animation. For a more complete description of configuration values, please see Appendix C.

Often we need to interrupt a running animation between two poses, and blend it to a different final pose. With the ANIMUS expression engine this process is very simple. Suppose we are currently turning to face the character mentioned before, while a new character suddenly touches Alebrije. We now want to stop paying attention to the first character and turn to see the second. Assume the weight for the correct gaze is 0.30 (a little bit to the left), and the appropriate pose is *gaze_short*:

```
1   // Get the current animation thread
2   CAnimationThread* pCurrentThread = getCurrentAnimationOrigin();
3
4   // Clear the animation queue, except for the current thread
5   clearAnimations(pCurrentThread);
6
7   // reset the animation engine states
8   setOptions(AMMSG_ENDSEQUENCE);
9
10  // redirect our thread to the new pose
11  pCurrentThread->redirect(pGazeShort,0.30f,fps,gps,0.5f,0.8f,ATHRD_FLEXIBLE|ATHRD_VIRTUAL);
12
13  // tell the animation engine to execute this thread
14  setKeyframe(pCurrentThread);
```

We are manually retrieving the currently active animation thread, modifying it, and putting int back into the animation engine. We can also pre–create threads and just pass them to the engine whenever we need them. Another common option is to *chain* a sequence of animations together to create a gesture[3]. This can be done directly in the physical layer definition file, or in realtime by placing a sequence of animation poses one after another.

[3]See Section 7.2.

Suppose we want Alebrije to wake up, go to the alert pose, gaze left, and go back to sleep:

```
1   // reset the animation engine states
2   setOptions(AMMSG_ENDSEQUENCE);
3
4   // first sequence: sleep->alert
5   setKeyframe(pSleep,0.0f,pAlert,0.0f,fps,gps,0.5f,0.0f,ATHREAD_BLENDED);
6
7   // second sequence: alert->gaze left
8   setKeyframe(NULL,0.0f,pGazeShort,0.0f,fps,gps,0.5f,0.0f,ATHREAD_BLENDED);
9
10  // final sequence: go back to sleep
11  setKeyframe(NULL,0.0f,pSleep,0.0f,fps,gps,0.5f,0.0f,ATHREAD_BLENDED);
```

Note that if we pass a *null* parameter as the first pose, ANIMUS will append it to the last known pose to create a chain.

Alebrije maintains a set of states to discriminate his current actions according to the last triggered reflex, and he is able to select animation sequences based on a hand coded verb graph similar to the one in Isla[16].

7.4 Summary and Discussion

We have stated the importance of a good animation engine, both artistically and technically. An adequate expression layer will secure the character's success, while a deficient animation engine ensures the character's failure, regardless of every other resource. We described the traditional methodologies for animation, being keyframed and interpolated motion the most widely used in domains like videogames. Verbs and adverbs animation systems have only begun to be applied to commercial domains, while pose–based and blend–based systems are still relatively new, mostly applied to experimental projects in arts and sciences only.

The concept of bones and skeletons for real time animation is also relatively new. It provides many important advantages over old keyframe–based systems, both in terms of storage (it is easier to keep a set of transformation matrices than an array of vertex positions) and performance (they allow the implementation of *skin* meshes). A current disadvantage is that acceleration hardware available today has a small set of registers, which limits the amount of bones that can be set at the same time in the register memory of the GPU (approximately 18). The alternative option is to apply bone transformations by software using the CPU, which implies a reduction in performance. This situation is bound to change with new hardware technologies and techniques that simplify and provide more resources for the use of programmable pipelines. The domain that normally pushes this technology is videogames, although new applications in the area of visualization technology and *new media arts* are recently requiring more power for their projects.

Chapter 8

A Minimalistic Implementation

"Distrust anyone who wants to teach you something"

-Peter J. Carroll

This chapter is a detailed description on to create the required C++ structures for a basic ANIMUS character. Each character is a self–contained object that inherits from the *CAnimusCharacter* class[1], which is a base class that provides the necessary functions to access all of the ANIMUS layer interfaces. The framework we are going to describe in this chapter provides no special functionality, but prepares all the resources and explains where to put specific code so that the user can create his/her own characters using ANIMUS.

The first step is to create two files, which for the sake of this example will be named *MyCharacter.h* and *MyCharacter.cpp*. Once this has been done, we need to include the following headers.

In MyCharacter.h:

```
1   #include "AnimusCharacter.h"          // main framework library
```

In MyCharacter.cpp:

```
1   #include "MyCharacter.h"              // our own header file
2   #include "VirtualWorld.h"             // includes methods for accessing the virtual world
```

The class declaration for our character will be named *CMyCharacter*. The following is a simple class skeleton.

[1] A complete layout of class relations and inheritance is described in Appendix A.

```
1   // Our character class definition
2   class CMyCharacter : public CAnimusCharacter
3   {
4     public:
5
6       CMyCharacter(void);
7       ~CMyCharacter(void);
8
9       // Initialization function
10      bool onCreate(void);              // Called when the user creates this character
11
12      // Callback function declarations
13      // ---------------------------------------
14
15      // General query functions
16      CStimuli*           query(CStimuli &s);   // query interface
17      CFeature*           query(FEATURE f);     // query interface
18      void*       publish();            // called by the virtual world blackboard
19
20      // Animus Interface
21      void onUpdatePerception();        // Called to update our perception layer
22      void onUpdateExpression();        // called to update our expression layer
23
24      // Called from the main application when we want to render our character.
25      void onRender(RENDERCONTEXT pRenderDevice, Matrix worldmat);
26
27      /// The messaging interfase
28      void onProcessMessage(MESSAGE aMessage);
29
30    private:
31
32      // Here we declare private members for our character. They can be  //
33      // several types of reflex agents, driver networks, or hight-level //
34      // perception elements. For the sake of this example, we are going //
35      // to keep a matrix to maintain this character's position, a       //
36      // pointer to a pose from the expression layer, and a driver to    //
37      // measure the 'boreness' of our character.
38
39      CPose*      m_pPose;          // pointer to some pose
40      CDriver*    m_pDriver;        // a driver to measure how bored we are
41      Matrix      m_matCharacter; // our 4x4 affine transformation matrix
42
43      // Also, keep a stimuli ready when the world manager asks to publish
44      // information.
45
46      CStimuli*   m_pPublishedStimuli;
47
48  }
49
```

At line 2, the class is created and inherited from CAnimusCharacter. Lines 6 and 7 show the conventional constructor and destructors. Line 10 declares a function that is used to initialize all the information required by the character, including member variables, states, and such. This function is automatically called by the framework when an instance of this character is created. Lines 16 and 17 create a query interface that is used by other characters when they want to ask our character about the value of a certain feature or stimuli. Line 18 is called by the virtual world manager when it needs to collect information about this character for the world blackboard.

Functions at lines 21 and 22 are also called by the framework every time the character is prompted to update its perception and expression systems, respectively. The first one will be called approximately 24 times per second, and the second will vary depending on the hardware capabilities of the

computer. When the situation is optimal, this will happen approximately 60 to 70 times per second. Line 26 declares the function that renders the geometry that was previously set in *onUpdateExpression*. The difference between updating the expression layer and rendering, is that in the first case, we change the geometry of the character by applying transformations (pose–blending), while the second takes the geometry and uses the graphic library to present it to the user. The last function at line 29 implements the message callback interface used by the ANIMUS layers to communicate.

The private part of the class corresponds to the internal member variables that will be utilized by our character to maintain its internal states, information, and processes. We can declare temporal memory objects, driver networks, or any other type of variable required by the character's design.

Now we are going to fill the body of each function. Let us begin with the constructor and destructor:

```
1   CMyCharacter::CMyCharacter(void)
2       : CAnimusCharacter()
3   {
4       m_pPose   = NULL;
5       m_pDriver = NULL;
6
7       m_matCharacter = IdentityMatrix();
8       m_pPublishedStimuli = new CStimuli();
9
10      // Any generic initialization of member variables goes here //
11  }
12
13  CMyCharacter::~CMyCharacter(void)
14  {
15     SAFE_DESTROY(m_pPose);
16     SAFE_DESTROY(m_pDriver);
17     SAFE_DESTROY(m_pPublishedStimuli);
18  };
```

The constructor will call the parent constructor, and perform conventional initialization of class member variables. Here we assume there exists a function called *IdentityMatrix()* that returns an identity $4x4$ matrix. The destructor will destroy any objects allocated in memory. In this case, we also assume there is a macro named *SAFE_DESTROY* that applies the *delete* operator to the provided object if it is not null. Next, we analyze the *onCreate(void)* function:

```
1   void CMyCharacter::onCreate(void)
2   {
3      // Initialize our pose
4      m_pPose = getAnimationManager()->getPoseByName(" somePose ");
5
6      // Initialize our driver and set the messaging system for it
7      m_pDriver = new CDriver("boreness",F_BORENESS, 0.0f, 0.0f, 5.0f, 10.0f,0);
8      m_pDriver->registerParent(this);
9
10     // Initialize our affine matrix
11     SetPosition( m_matCharacter, 0.5f, 0.0f, 7.0f );
12
13     // Here we initialize any other required variable //
14     return true;
15  }
```

Line 4 reveals that we can access the expression layer through the *getAnimationManager* function provided by *CAnimusCharacter*. Line 7 initializes our driver and registers this object to receive any message it generates. A driver will send messages to all objects registered with the *registerParent* function. In this case, we will be automatically notified by a call to *onProcessMessage* when the magnitude of our driver surpasses its threshold, defined initially as 10. Finally, line 11 assumes there exists a function named *SetPosition* that modifies our affine matrix to set ourselves at $x = 0.5, y = 0.0, z = 7.0$. At the end, we just return *true*, or *false* if some initialization parameter failed to be set.

The query interface functions are declared as follows:

```
1  CStimuli* CMyCharacter::query(CStimuli &s)
2  {
3     return 0;
4  }
5
6  CFeature* CMyCharacter::query(FEATURE f)
7  {
8     return 0;
9  }
10
11 void* CMyCharacter::publish()
12 {
13    // publish our position
14    CFeature position(POS,GetPosition(m_matCharacter));
15    m_pPublishedStimuli->setFeature(position);
16
17    return (void*)m_pPublishedStimuli;
18 }
```

Both declarations for *query* return null, which means this character will not respond queries from other characters. If desired, we could analyze each query and respond with appropriate information. Function *publish*, on the contrary, will reveal our position to the virtual world's blackboard (assuming the function *GetPosition* returns a 3D vector obtained from our affine matrix), so that other characters will be able to know where we are.

The update functions are declared as follows:

```
1  void CMyCharacter::onUpdatePerception()
2  {
3     // get a pointer to the blackboard
4     LPBLACKBOARDITERATOR pTheBlackboard = getVirtualWorld()->getBlackboard();
5
6     // If we sense other character, we are amused. Otherwise, increase our
7     // boreness ratio
8     if(!pTheBlackboard)
9        *m_pDriver+=borenessIncreaseFunction();
10    else
11       *m_pDriver-=borenessDecreaseFunction();
12 }
13
14 void CMyCharacter::onUpdateExpression()
15 {
16    // just tick our geometry
17    tickArticulatedCharacter();
18 }
```

The *onUpdatePerception* function creates a pointer to the blackboard by using the *getVirtualWord* function declared in *CAnimusCharacter*. We then increase or decrease our boreness ratio depending on whether we find other characters or not. The functions *borenessIncreaseFunction* and *borenessDecreaseFunction* could become something more interesting and consider stimuli provided by existing characters, analyze their identities, or any other desired action. Function *tickArticulatedCharacter* at *onUpdateExpression* will just tick the expression layer so that it blends and transforms the character's geometry as required.

The rendering function can be something like this:

```
1   void CMyCharacter::onRender(RENDERCONTEXT pRenderContext, Matrix worldMat)
2   {
3      // Multiply our matrix by the world matrix
4      Matrix currentMatrix = MatrixMultiply(worldMat, m_matCharacter);
5
6      // use the render context to render our geometry
7      render( pRenderContext, currentMatrix );
8   }
```

In this function, we combine the current world matrix and our own matrix into a single affine transformation matrix, which is provided to the render function declared in *CArticulatedCharacter*, and triggers the skin geometry rendering functions.

Finally, we declare our message processing function:

```
1   void CMyCharacter::onProcessMessage(MESSAGE aMessage)
2   {
3      // deal with messages coming from our driver
4      if(aMessage->referer() == (void*)m_pDriver)
5      {
6         // it is not necessary to analyze the message itself, since
7         // we are just expecting to be called for DM_THRESHOLD (driver
8         // magnitude has exceeded the threshold)
9         performSleepingAnimation();
10     }
11  }
```

Since most framework messages will be directly handled by CAnimusCharacter, we just need to concern ourselves with messages coming from objects we created. In this case, we instructed our driver to call us when its threshold reached 10, so if we are at this function, it means the character is bored. In that case, we call *performSleepingAnimation*, which could probably trigger an animation that shows how bored we are. Depending on how we configure our animation engine, we can also get a message when such animation has ended, so that we can take the pertinent actions (probably begin to snore, or shut down our perception system).

Finally, whenever we want to create a new instance of CMyCharacter, the creation routine will go as follows:

```
1   // ... somewhere in our program, we decide to create a CMyCharacter instance ...//
2   CMyCharacter *newCharacter = new CMyCharacter();
3
4   // Create the character, giving the location of the required initialization files
5   newCharacter->Create("myCharacterDefinitions.ini");
```

The *Create* function expects a *character definition file*, which is just a better way to specify the required files for creating this character. We previously mentioned that there are several xml definition languages used to configure ANIMUS layers. To provide their names and the magnitudes of current and future configuration values in one single step at runtime, ANIMUS reads all the required initialization information from a file. In our case, such file would look like this:

```
1   ########################################
2   # This is an example character definition
3   # file for CMyCharacter
4   ########################################
5
6   # Information for the perception layer
7   perceptionNetwork myChacter_perception_network.xml
8
9   # Information for the expression layer
10  meshFile myCharacter_mesh.def
11  poseFile myCharacter_pose.xml
```

Line 7 specifies the xml file with configuration information for the perception network. Line 10 tells ANIMUS about the location of the file containing our character's 3D model and bone structure, and line 11 names the physical layer definition file to use. A great advantage of using definition files is that we can swap or test different geometry models and configurations in realtime, without having to recompile our whole program.

8.1 Conclusions

We have described a minimalistic code skeleton for the implementation of an ANIMUS character. Provided that we have 3D geometry and at least two poses for the *boring* animation mentioned in this chapter, the character is fully functional and will play an animation when no more characters are in sight. Creating believable and interesting behavior is of course more complicated than that, but ANIMUS provides the necessary tools for the artists and designers to go as far as they need.

As the reader might have already noted, the CAnimusCharacter base class encapsulates most of the execution logic, so completing the character is just a matter of providing code for several pure functions that respond to specific events, plus adding particular functions concerning the unique design of our character. This scheme greatly simplifies the task of creating new characters, allowing the designer to invest time into the creation of the character's behavior. For very specific functionality, or a better control over the framework, the reader will find useful information in the source code header files.

Chapter 9

General Comments and Future Work

There is often a clear distinction between the artist's and the scientist's way of thinking, and there is also the people that have a little bit of both worlds and sometimes don't quite fit into one without including the other. Sometimes, great and unsuspected things happen when art and technology work together. The result can be often seen in the field of new media arts, film, videogames, and many other forms of expression (artistic and technological). However, putting artists and scientists to work together presents two difficulties:

- Artists do not understand the language of science, and *viceversa*
- Scientists sometimes believe they have no need for art, and *viceversa*

This barrier between art and science is fading away with surprising speed, in fact, we are suddenly presented with the opportunity to join these disciplines together, but lack the appropriate tools and work methodologies for doing so.

The ANIMUS framework is an attempt to construct a next generation tool designed to deal with the two previously mentioned problems, bringing artists and scientists together, and satisfying the needs of each one in their own fields. Creating synthetic characters is an excellent place to do this: both art and science can provide their respective cutting–edge resources to obtain innovative results.

The expression layer is probably the most immediately useful tool from the framework. It is also the one that can be more easily automatized since one can almost directly go from a 3D model in a modeling package to a fully configured animation engine, ready to be applied to our characters. The cognition layer lacks the complete functionality of the ART network, but it already provides enough methods to sustain a working character and accomplish most of its design objectives.

The modularity shown in the ANIMUS framework results useful for real–life applications because in most projects, people working with artistic design,

application logic, animation, and graphic rendering, are different work groups. Because ANIMUS provides integration by a set of transparent non–coupled interfaces, the framework effectively allows such groups to work independently and join their work at any time.

Much of the work is incomplete. Although the perception and expression layers have reached a mature state, the foundations of the cognition layer are just sketched by a few classes and objects. This is due to the fact that designing a model for social interaction, emotional behavior, memory, decision making and other high–level activities worths, by itself, years of investigation and research. Nevertheless, the ANIMUS framework will constitute an essential starting point for what is to come. Without a framework like this, our models of emotions and high–level behavior will not be more than spirits without a body.

Future work will be essentially centered on the cognition layer. The next step is to design the *mind* and *spirit* that will inhabit the body of our ANIMUS characters. Specifically containing some of the following features:

- A High–level model for the temporal representation of the world. The temporal memory module will be the main element of a new module that will discriminate and record information related to the character and other elements in the world through the duration of a certain time frame. Recorded elements will be provided by internal and external perception.
- A knowledge acquisition model that will apply analysis techniques over the information in the temporal world representation model. The intention is to create an automated module that selects specific information to be *remembered* in memory models without temporal frames. The criteria for selecting perceived information, represent knowledge and construct symbolic instances of information is still to be defined.
- A model for the construction, maintainance and execution of plans, probably similar to the work of Gratch[23], and strongly based on concepts from Gordon[22, 19, 20].
- A model of emotions based on hand–coded constraints (according to the artistic concept of our characters) and the relationship between the character's plans, priorities, and particular emotional features as described in Scott[57]. Emotions generated from this model will be shown by the expression layer, and bias the functionality of every other model in the cognition and perception layers.
- A model of social behavior and interaction.

Additionally, we have plans for implementing high–level extensions for the perception layer (including task–dependent attention, similar to Laar[54]).

A great number of applications will result from this, including a new generation of characters that will make immersive environments a more realistic experience, and better agents for support applications, research, and entertainment.

Bibliography

[1] Francisco J.Seron Alfredo Pina and Diego Gutierrez. The alvw system: an interface for smart behavior–based 3d computer animation. *ACM Smart Graphics Symposium*, 2002.

[2] Elisabeth André et al. *Integrating Models of Personality and Emotions into Lifelike Characters.* In: Affect in Interaction, Springer, 2000.

[3] G.L. Clore Andrew Ortony and A. Collins. *The Cognitive Structure of Emotions.* Cambridge University Press, 1988.

[4] Jianchang Mao Anil K. Jain and K. Mohiuddin. Artificial neural networks: A tutorial. *IEEE Computer Special Issue on Neural Computing*, March 1996.

[5] I. Zukerman Ann E. Nicholson and C.D. Oliver. Towards a society of affect–driven agents. *In Proceedings of COGSCI98*, pages 776–781, August 1998.

[6] Electronic Arts. The sims©.

[7] Guszti Bartfai. An art–based modular architecture for learning hierarchical clusterings. *Neurocomputing*, 13:31–45, 1996.

[8] Robert Brooks and Lynn A. Stein. Building brains for bodies. *Proceedings of the AIAA/NASA Conference on Intelligent Robots for Factory, Field, Service, and Space*, March 1994.

[9] Robert Brooks Brooks. A robot that walks; emergent behavior from a carefully evolved network. *Neural Computation, 1:2*, pages pp. 253–262, Summer 1989.

[10] Nancy Cantor and John Kihlstrom. *Personality and Social Intelligence.* Century Psychology Series, 1987.

[11] G. Carpenter and S Grossberg. Adaptive resonance theory (art). *In The Handbook of Brain Theory and Neural Networks, MIT Press*, pages 79–85, 1995.

[12] Bobby Bodenheimer Charles Rose and Michael F. Cohen. Verbs and adverbs: Multidimensional motion interpolation using radial basis functions. *IEEE Computer Graphics and Applications*, 18(5):32–40, 1998.

[13] R. D. Terzopoulos, X. Tu. Artificial fishes: Autonomous locomotion, perception, behavior and learning in a simulated physical world. *Artificial Life*, 1(4):327–351, December 1994.

[14] Antonio R. Damasio. *Descarte's Error: Emotion, Reason and the Human Brain.* Avon Books, 1995.

[15] Antonio R. Damasio and Hanna Damasio. Cortical systems for retrieval of concrete knowledge: The convergence zone framework. *Large-Scale Theories of the Brain, MIT Press*, pages 61–74, 1995.

[16] Marc Downie Damian Isla, Robert Burke and Bruce Blumberg. A layered brain architecture for synthetic creatures. *International Joint Conferences on Artificial Intelligence*, August 2001.

[17] Discreet. 3dstudio max. 3d modeling and animation package.

[18] Mark Downie. Animation, music: The music and movement of synthetic characters. Master's thesis, Media Arts and Sciences, MIT, 2001.

[19] Andrew S. Gordon. Playing chess with machiavelli: Improving interactive entertainment with explicit strategies. *2001 AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, 2001.

[20] Andrew S. Gordon. The representational requirements of strategic planning. *Fifth symposium on Logical Formalizations of Commonsense Reasoning*, 2001.

[21] Andrew S. Gordon. Strategies in analogous planning cases. *Proceedings of the 23rd Annual Conference of the Cognitive Science Society*, 2001.

[22] Andrew S. Gordon. The theory of mind in strategy representations. *In Proceedings of the 24th Annual Conference of the Cognitive Science Society*, August 2002.

[23] Jonathan Gratch. Modeling the interplay between emotion and decision-making. *Ninth Conference on Computer Generated Forces and Behavioral Representation*, 2000.

[24] Jonathan Gratch and Stacy Marsella. Modeling emotions in the mission rehearsal exercise. *10th Conference on Computer Generated Forces and Behavioral Representation*, May 2001.

[25] B. Hayes-Roth. An architecture for adaptive intelligent systems. *Artificial Intelligence: Special Issue on Agents and Interactivity*, 17:329–365, 1995.

[26] http://www.alicebot.org. Alice: Artificial linguistic computer entity.

[27] http://www.h anim.org. H—anim, humanoid animation working group.

[28] Damian Isla and Bruce Blumberg. New challenges for character-based ai for games. *AAAI Spring Symposium on AI and Interactive Entertainment*, March 2002.

[29] Robert Burke. Damian Isla. Marc Downie. Yuri Ivanov and Bruce Blumberg. Creature smarts: The art and architecture of a virtual brain. *Game Developers Conference*, 2001.

[30] Xiaoyuan Tu John Funge and Demetri Terzopoulos. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. *ACM Computer Graphics Procedures*, 1999.

[31] A. Bryan Loyall Joseph Bates and W. Scott Reilly. Integrating reactivity, goals and emotions in a broad agent. *14th Annual Conference of the Cognitive Science Society*, July 1992.

[32] Bryan Loyall Joseph Bates and W. Scott Reilly. An architecture for action, emotion and social behavior. Technical report, School of Computer Science, Carnegie Mellon University, 1992.

[33] A. Bruderlin K. Amaya and T. Calvert. Emotion from motion. *Graphics Interface '96*, pages 222–229, May 1996.

[34] Christopher Kline and Bruce Blumberg. The art and science of synthetic character design. *Proceedings of the AISB Symposium on AI and Creativity in Entertainment and Visual Art*, 1999.

[35] Christof Koch and Francis Crick. Some further ideas regarding the neuronal basis of awareness. *Large–Scale Theories of the Brain, MIT Press*, pages 93–109, 1995.

[36] Barry Kort and Rob Reilly. Analytical models of emotions, learning and relationships: Towards an affect–sensitive cognitive machine. *Proceedings of the International Conference on Virtual Worlds and Simulations*, 2003.

[37] Barbara Krahe. *Situation Cognition and Coherence in Personality.* Cambridge University Press, 1990.

[38] Robert Brooks Liana M. Lorigo and W. E. Grimson. Visually-guided obstacle avoidance in unstructured environments. *Proceedings of IROS '97*, pages 373–379, September 1997.

[39] A. Bryan Loyall and Joseph Bates. Personality–rich believable agents that use language. *Proceedings of the First International Conference on Autonomous Agents*, February 1997.

[40] Oscar Gonzalez Loyo. Karmatron y los transformables. *Editorial Antea*, 1999.

[41] Fred Charles Marc Cavazza and Steven J. Meads. Intelligent virtual actors that plan...to fail. *In proceedings of the Third International Symposium On Smart Graphics*, March 2003.

[42] Stacy Marsella and Jonathan Gratch. Modeling the interplay of emotions and plans in multi–agent simulations. *In proceedings of the 23rd Annual Conference of the Cognitive Science Society*, 2001.

[43] Stacy Marsella and Jonathan Gratch. A step toward irrationality: Using emotion to change belief. *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems*, 2002.

[44] Michael Mateas. *An Oz–centric Review of Interactive Drama and Believable Agents.* AI Today: Recent Trends and Developments. Lecture Notes in AI 1600. Berlin. Springer, 1999.

[45] Richard V. Ivry Michael Gazzaniga and George Mangun. *Cognitive Neuroscience, The Biology of The Mind.* Norton, New York, 1998.

[46] Marc Cavazza Miguel Lozano, Steven J. Mead and Fred Charles. Search–based planning for character animation. *In Proceedings, ADCOG, Hong Kong*, 2002.

[47] D Mumford. Neuronal architectures for pattern-theoretic problems. *Large–Scale Theories of the Brain, MIT Press*, pages 125–152, 1995.

[48] Ommanney and Schanker. *The Stage and the School.* McGraw-Hill, New York, 1972.

[49] M. Green Pablo Figueroa and H. J. Hoover. Intml: A description language for vr applications. *Web3D'02*, February 2002.

[50] Maria Miceli Paola Rizzo, Manuela Veloso and Amedeo Cesta. Personality–driven social behaviors in believable agents. *AAAI Fall Symposium on Socially Intelligent Agents*, 1997.

[51] Olivier Parisy and Christophe Schlick. A physically realistic framework for the generation of high–level animation controllers. *International Symposium on Smart Graphics*, 2002.

[52] Olivier Parisy and Christophe Schlick. A physically realistic framework for the generation of high–level animation controllers. *In proceedings of the Second International Symposium On Smart Graphics*, june 2002.

[53] Bruce Blumberg Pattie Maes, Trevor Darrell and Alex Pentland. The alive system: Wireless, full–body interaction with autonomous agents. *Special Issue on Multimedia and Multisensory Virtual Worlds, ACM Multimedia Systems*, 1996.

[54] Tom Heskes Pierre Van de Laar and Stan Guielden. Task–dependent learning of attention. *Neural networks*, pages 981–992, 1997.

[55] Robert Plutchik and Hope R. Conte. *Circumplex Models of Personality and Emotions.* American Psychological Association, 1997.

[56] V. S. Ramachandran and Sandra Blakeslee. *Phantoms in The Brain.* Quill, 1999.

[57] W. Scott Neal Reilly. *Believable Social and Emotional Agents.* PhD thesis, Carnegie Mellon University, 1996.

[58] Picard Rosalind W. Toward agents that recognize emotion. *Actes Proceedings IMAGINA*, 1998.

[59] Picard Rosalind W and Jonathan Klein. Computers that recognize and respond to user emotion: Theoretical and practical implementations. Technical report, MIT Media Laboratory, 2000.

[60] Elias Vyzas Rosalind W. Picard and Jennifer Healey. Toward machine emotional intelligence: Analysis of affective physiological state. Technical report, MIT Media Laboratory, 1992.

[61] Charles Rose. *Multidimensional motion interpolation using radial basis functions.* PhD thesis, Princeton, 1999.

[62] Daniel Rousseau and Barbara Hayes-Roth. Improvisational synthetic actors with flexible personalities. Technical report, Stanford University, 1997.

[63] Jefferson A. Singer and Peter Salovey. *The Remembered Self.* The Free Press, 1993.

[64] Nancy L. Stein and Keith Oatley. *Basic Emotions.* Lawrence Erlbaum Associates, Publishers, 1992.

[65] I.J. Taylor. A self adapting parallel implementation of the art 2–a neural network. *Internal paper, University of Walles, College of Cardiff*, 1995.

[66] Demetri Terzopoulos and Keith Waters. Analysis and synthesis of facial image sequences using physical and anatomical models. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 15, 1993.

[67] http://www.blender3d.org The Blender Foundation. Blender.

[68] Frank Thomas and Ollie Johnston. *Disney Animation: The Illusion of Life.* Abbeville Press, New York, 1981.

[69] Daniel Torres and Pierre Boulanger. A perception and selective attention system for synthetic creatures. *In proceedings of the Third International Symposium On Smart Graphics*, March 2003.

[70] Daniel Torres and Pierre Boulanger. The animus project: A framework for the creation of synthetic characters on immersed environments. *To appear on Proceedings for ACM VRST*, 2004.

[71] Juan D. Velázquez. When robots weep: Emotional memories and decision–making. *AAAI Proceedings*, 1998.

[72] Stella Vosniadou and Andrew Ortony. *Similarity and Analogical Reasoning.* Cambridge University Press, 1989.

[73] Joseph Bates W. Scott Reilly. Natural negotiation for believable agents. *Tech. Report CS-95-164, Carnegie Mellon University*, 1995.

[74] Alias Wavefront. Maya.

[75] D. J. Wiley and J.K. Hahn. Interpolation synthesis for articulated figure motion. *In Proceedings of the Virtual Reality Annual International Symposium*, pages 157–160, March 1997.

[76] L. Williams and A. Bruderlin. Motion signal processing. *Computer Graphics, Proceedings of SIGGRAPH 95*, pages 97–104, August 1995.

Part III

Appendices

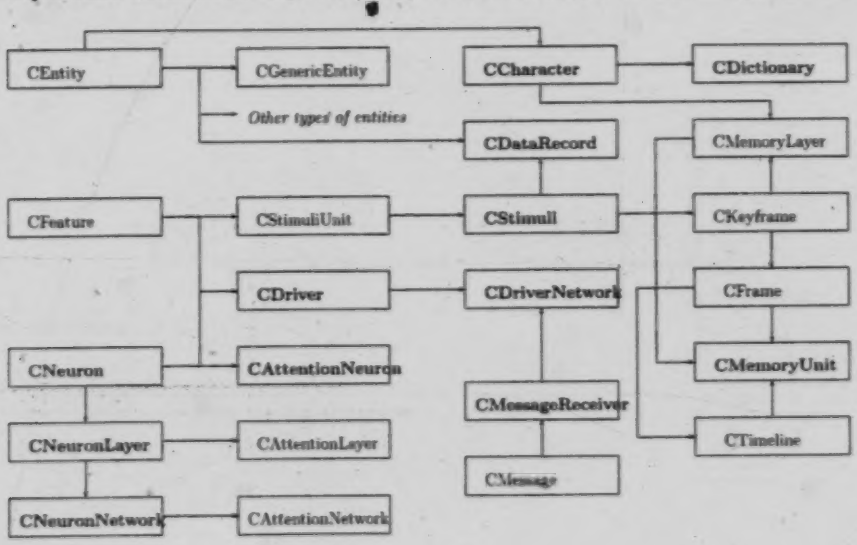
Appendix A

Diagram of ANIMUS Classes

This appendix contains diagrams that illustrate the relationship between the main classes that compose the ANIMUS framework. Each class is contained inside its own header definition file. For more information and implementation details, please refer to the code documentation (not provided in this thesis) or the notes and comments on the code files themselves.

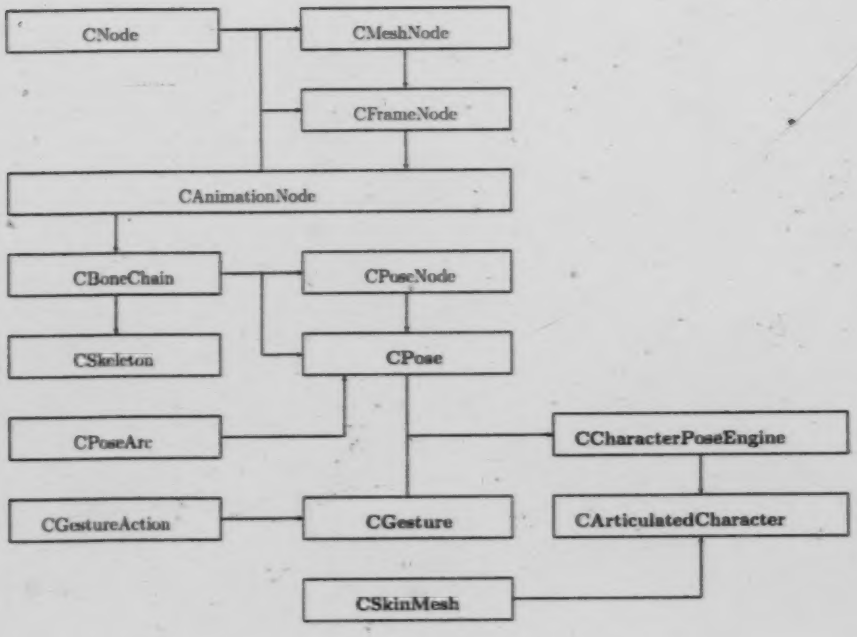
A.1 ANIMUS Perception and Cognition Classes

The following diagram shows the class layout for the Perception Layer. CLayer, CNeuronLayer, and CNeuronNetwork, can be used as base classes for implementing different types of AI neural algoritms. CEntity can also be used as a base class for creating particular entity types, like CGenericEntity in the diagram. The directional arrows imply inheritance or the administration of the preceding class by the pointed class. Classes with names in bold represent important definitions that are likely to be directly used by the programmer.



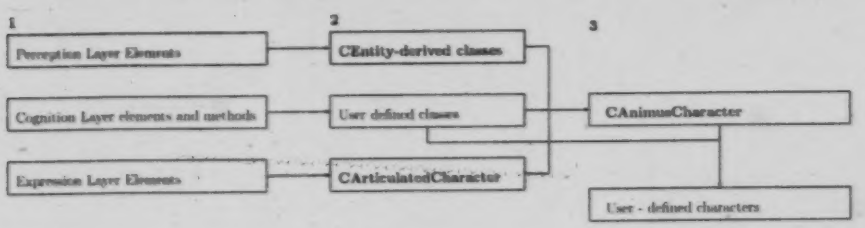
A.2 ANIMUS Expression Classes

The following diagram illustrates the relationship between the classes composing the ANIMUS pose–based animation engine. CSkinMesh is the entry point for the raw geometric information (a file containing vertices, bones, index and face lists, and such). Remember that although these diagrams show the relationship between classes, not all relations are inheritance–related. Please refer to the code documentation for further details.



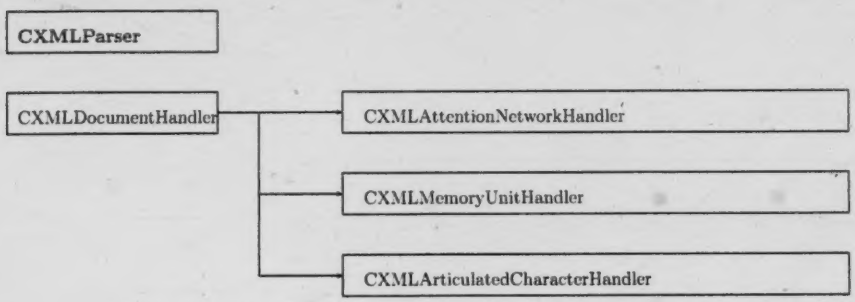
A.3 Abstraction Levels

The next diagram shows three different levels of abstraction, being **1** the lowest and **3** the highest. At level 1 we have all the basic classes for the different ANIMUS layers. At level 2 we have high–level administration classes and user–defined classes. At level 3 we have the master class, *CAnimusCharacter*, which provides methods for accessing all three layers. Usually, classes defined and derived by the user are also located at this level, unless we want to make important changes to the functionality of the ANIMUS base classes.



A.4 XML Parsing objects and its derivates

ANIMUS contains two main classes designed to read XML data from files. For information on how to use these files, please refer to the code documentation. The next diagram shows the currently available elements:



Appendix B

DTD's for the ANIMUS Framework Files

This appendix describes the XML specifications of several definition languages created to design ANIMUS components. Each section begins with a brief description and application of the language, the corresponding DTD, and a small example.

B.1 Networks

Networks are composed by a set of layers, which are themselves containers for individual nodes. The network currently implemented in the ANIMUS framework is called *Attention Network* and its described in Chapters 5 and 8. The following definition language has been designed to structure simple networks for general use. The programmer can take advantage of the language to define not only attention networks, but other network–based components for his/her own virtual characters.

B.1.1 Network DTD

```
<!ELEMENT attentionNetwork (layer)+>
<!ELEMENT layer (neuron)+>
<!ELEMENT neuron (arc)*>
   <!ATTLIST neuron ft CDATA "F_UNKNOWN">
   <!ATTLIST neuron dc CDATA "60">
   <!ATTLIST neuron w  CDATA "1.0000">
   <!ATTLIST neuron u  CDATA "0.5000">
   <!ATTLIST neuron t  CDATA "0.0000">
   <!ATTLIST neuron mt CDATA "0.0000">
   <!ATTLIST neuron cw (true|false) "false">
   <!ATTLIST neuron cu (true|false) "false">
   <!ATTLIST neuron outp (true|false) "false">
<!ELEMENT arc EMPTY>
   <!ATTLIST arc prev CDATA #REQUIRED>
   <!ATTLIST arc w    CDATA "1.0000">
```

B.1.2 Parameters

- **attentionNetwork** begins the definition of the network
- **layer** begins the definition of a layer
- **neuron** begins the definition of a neuron
- **ft** uniquely identifies the feature perceived by a neuron
- **dc** specifies neuron decay in frames
- **w** specifies neuron weight
- **u** indicates learning (or adaptation) factor
- **t** the threshold element of the network
- **mt** maximum threshold value
- **cw** constant weight indicator
- **cu** constant learning indicator
- **outp** when true, this network provides output to the resultant vector of the layer
- **arc** begins an arc
- **prev** neuron in the previous layer connected to this neuron
- **w** weight that modifies the output of the previous neuron to constitute the input to a new neuron

B.1.3 Example

```
<attentionNetwork>
   <layer>
      <neuron ft='F_VELOCITY' dc='30' w='1.0' u='0.2' t='0.4' mt='0.02' outp='true'></neuron>
   </layer>
   <layer>
      <neuron ft='FCUSTOM_1' dc='15' w='1.0' u='0.5' t='0.02' mt='0.5' outp='true'>
         <arc prev='F_VELOCITY' w='2.0000'/>
      </neuron>
   </layer>
</attentionNetwork>
```

B.2 Physical Layer Definition Files

Contain information to help the expression layer provide a transparent level of abstraction over the character's geometry. These files can divide the skeleton of a single character into several bone chains, specify nodes and weighted poses, and construct precoded animation sequences called *gestures*. For a more complete description on how to use these files and the expression layer, please refer to Chapters 7 and 8.

B.2.1 Physical Layer DTD

```
<!ELEMENT articulatedCharacter (chain+,pose+,gesture+)>
   <!ATTLIST articulatedCharacter ipose IDREF #REQUIRED>
<!ELEMENT chain EMPTY>
   <!ATTLIST chain id    ID    #REQUIRED>
   <!ATTLIST chain bones CDATA #REQUIRED>
<!ELEMENT pose ((node+|skeleton+),arc*)>
   <!ATTLIST pose pid ID #REQUIRED>
<!ELEMENT node (skeleton+)>
   <!ATTLIST node w CDATA "0.0000">
<!ELEMENT skeleton EMPTY>
   <!ATTLIST skeleton at    CDATA #REQUIRED>
   <!ATTLIST skeleton set   IDREF #REQUIRED>
<!ELEMENT arc EMPTY>
   <!ATTLIST arc to   IDREF #REQUIRED>
   <!ATTLIST arc dist CDATA #REQUIRED>
   <!ATTLIST arc type (blend|sequential) "blend">
<!ELEMENT gesture (concurrent|walkto)+>
   <!ATTLIST gesture gid ID #REQUIRED>
   <!ATTLIST gesture fps CDATA "24">
   <!ATTLIST gesture events  (none|repeat) "none">
<!ELEMENT concurrent (walkto+)>
<!ELEMENT walkto EMPTY>
   <!ATTLIST walkto pid IDREF #REQUIRED>
   <!ATTLIST walkto sec CDATA #REQUIRED>
   <!ATTLIST walkto choose (strict|closest|endeffect) "closest">
   <!ATTLIST walkto weight CDATA "0.0000">
   <!ATTLIST walkto wait (true|false) "true">
   <!ATTLIST walkto easing CDATA "0">
```

B.2.2 Parameters

- **articulatedCharacter** begins an articulated character file
- **ipose** points to the id of the default pose
- **chain** defines a bone chain
- **id** the unique id of a bone chain
- **bones** a list of bone id's separated by commas
- **pose** begins a pose structure
- **pid** unique id for a pose
- **node** begins a node structure
- **w** weight assigned to a particular node
- **skeleton** begins a skeleton structure
- **at** indicates a snapshot of keyframe in the character's geometry file for the set of bones assigned to a pose
- **set** the unique id of a bone chain that must be animated to a certain keyframe
- **arc** begins a link between two poses

- **to** id reference of the destination pose
- **dist** assigned distance for the arc between two poses
- **type** the type of arc. Blend means we must interpolate from one pose to the next one. Sequential means there are n keyframes between the first and last poses, and the animation must iteratively interpolate between them.
- **gesture** begins a gesture structure
- **gid** unique id to identify a gesture
- **fps** frames per second when animating this gesture
- **events** specifies extra information about a gesture. Not very useful now.
- **concurrent** begins a concurrent animation inside a gesture
- **walkto** begins a sequential animation inside a gesture
- **pid** id of the pose we are blending to
- **sec** duration of an animation segment in seconds
- **choose** different type of controllers for selecting an appropriate weight value for the second pose. Not yet supported.
- **weight** weight for the destination pose
- **wait** indicates whether an animation must start immediately or wait for a previous one to end first
- **easing** float for specifying ease–in (-1) or ease–out (1) behavior for the animation

B.2.3 Example

```
<!-- this would be our initial position. Also, it tells  -->
<!-- the arm to rest as indicated in keyframe 1           -->
<pose pid='resting_position'>
   <skeleton at='1' bones='shoulder,elbow,wrist'/>
   <arc to='hand_wave' dist='5'/>
</pose>

<!-- This pose has a fuzzy range of positions depending  -->
<!-- on the indicated weight. If we indicate a high      -->
<!-- weight, the head of the character also moves.       -->
<pose pid='hand_wave'>
   <node w='0'>
      <skeleton at='2' bones='shoulder,elbow,wrist'/>
   </node>
   <node w='100'>
      <skeleton at='3' bones='shoulder,elbow,wrist'/>
      <skeleton at='8' bones='neck'/>
   </node>
   <arc to='resting_position' dist='5'/>
   </pose>
```

```
<!------------- gestures ------------- >

<!-- a default hand wave. It goes to resting position   -->
<!-- first, then chooses the closest hand wave and      -->
<!-- returns to the resting position. If it must exit   -->
<!-- that is done at resting position.                  -->
<gesture gid='default_wave' fps='32'>
   <walkTo pid='resting_position' sec='0.3' wait='true' />
   <walkTo pid='hand_wave' sec='1' wait='true' easing='1'/>
   <walkTo pid='resting_position' sec='1' wait='false' />
</gesture>

<!-- Similar gesture, but we wave in admiration     -->
<!-- difference is in second walkTo                 -->
<gesture gid='admiration_wave' fps='32'>
   <walkTo pid='resting_position' sec='0.3' wait='true'/>
   <walkTo pid='hand_wave' sec='1' choose='strict' weight='1' wait='true' easing='1'/>
   <walkTo pid='resting_position' sec='1' wait='false' />
</gesture>

<!-- a cycle gesture. assume all indicated poses exist -->
<!-- the second pose depends on how far we go. All the -->
<!-- collection of walking keyframes are contained in  -->
<!-- 'various_step_ranges' and we choose the one whose -->
<!-- weight is closest to the distance we want to      -->
<!-- travel. In a cat, for example, longer distances   -->
<!-- might require jumps, while short dist. require    -->
<!-- gaits. This gesture can only be exited at the last-->
<!-- step.                                             -->
<gesture gid='default_walk' fps='32' events='repeat'>
   <walkTo pid='initial_walk' sec='0.2' wait='true'/>
   <walkTo pid='various_step_ranges' sec='1' choose='strict' weight='$distance' wait='true'/>
   <walkTo pid='initial_walk' sec='1' wait='false'/>
</gesture>

</articulatedCharacter>
```

Appendix C

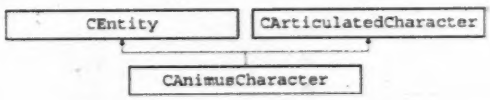
Framework Objects and Functions

This chapter describes some of the main classes of the ANIMUS framework. Documentation generated by *doxygen*[1].

C.1 CAnimusCharacter Class Reference

`#include <AnimusCharacter.h>`

Inheritance diagram for CAnimusCharacter::



C.1.1 Detailed Description

This is probably the most important class of the project.

It conjugates the 3 big layers into one single abstraction from which special characters can be designed.

The class provides basic functionallity. It must be derived to create each specific character.

For complete documentation of this class, look at section 10 in the conventions.txt document.

[1]Doxygen Version 1.3.3; Dimitri van Heesch

C.1.2 Constructor & Destructor Documentation

C.1.2.1 CAnimusCharacter::CAnimusCharacter (void)

default constructor

C.1.2.2 CAnimusCharacter::~CAnimusCharacter (void)

destructor

C.1.3 Member Function Documentation

C.1.3.1 bool CAnimusCharacter::create (float *pFps*, float *gFps*, char * *pDefinitionFile*, CExceptionManager * *pExceptionManager*, LPDIRECT3DDEVICE8 *pd3ddevice*)

load everything from a single file.

This option is highly recommended. pFps: number of perception samples per second gFps: graphic engine's average fps pDefinitionFile : a file containing the required definition parameters

C.1.3.2 bool CAnimusCharacter::loadExpressionLayer (char * *xFile*, char * *poseFile*, LPDIRECT3DDEVICE8 *pd3ddevice*)

The expression layer constructor.

Contains the name of the file that has the information for the expression elements

C.1.3.3 bool CAnimusCharacter::loadPerceptionLayer (char * *uid*, ETYPE *type*, char * *perceptionFile*)

The perception layer constructor.

It contains the name of the xml file with the definitions for the perception network

C.1.3.4 virtual bool CAnimusCharacter::onCreate (void) [pure virtual]

called when the application uses the create(...) function.

This is done so that the child class doesn't have to override such function and also provides means of a more organized workflow. Here the child does any extra initialization

C.1.3.5 void CAnimusCharacter::onMessage ()

We implement the **onMessage()**(p. 105) callback so that we manage the message queue and the child class just handles the received messages with onReceivedMessage().

Reimplemented from **CArticulatedCharacter** (p. 114).

C.1.3.6 virtual void CAnimusCharacter::onProcessMessage (MESSAGE *aMessage*) [pure virtual]

called once for each message received

C.1.3.7 virtual void CAnimusCharacter::onRender (LPDIRECT3DDEVICE8 *pd3dDevice*, LPD3DXMATRIX *pworldmat*) [pure virtual]

When we are called to render.

Usually we just call the render member of the expression layer

C.1.3.8 void CAnimusCharacter::onUpdate (float *eTime*)

When we are called to update.

The parameter is the time elapsed since the last call. This function triggers other callback functions like onUpdatePerception and onUpdateExpression

C.1.3.9 virtual void CAnimusCharacter::onUpdateExpression () [pure virtual]

This calls the character to update its expression layer.

C.1.3.10 virtual void CAnimusCharacter::onUpdatePerception () [pure virtual]

This calls the character to update its perception layer.

C.1.3.11 void CAnimusCharacter::setVirtualWorld (class CVirtualWorld * *pVirtualworld*)

sets the virtual world pointer

C.1.4 Field Documentation

C.1.4.1 D3DXMATRIX CAnimusCharacter::m_characterMatrix [protected]

Our very own world matrix. Useful when rendering the character.

C.1.4.2 CDictionary CAnimusCharacter::m_dictionary [protected]

a dictionary of known entities

C.1.4.3 float CAnimusCharacter::m_fAccTime [protected]

accumulated time

C.1.4.4 float CAnimusCharacter::m_fGFPS [protected]

the graphic engine's fps.
We get this value from the application configuration file

C.1.4.5 float CAnimusCharacter::m_fPerceptionDelta [protected]

how often do we update based on how many times per second are we required to sample our environment?

C.1.4.6 CAttentionNetwork* CAnimusCharacter::m_pAttentionNetwork [protected]

The attention network of this character.

C.1.4.7 CDriverNetwork* CAnimusCharacter::m_pBasicDrivers [protected]

a basic driver network for our character

C.1.4.8 CExceptionManager* CAnimusCharacter::m_pExceptionManager [protected]

a variable for output of errors

C.1.4.9 CStimuli* CAnimusCharacter::m_pProperties [protected]

This is our string of properties.
It is the responsibility of the derived class to fill it with significant information

C.1.4.10 CStimuli* CAnimusCharacter::m_pPublicStimuli [protected]

This is what other agents can discover about ourselves at time t.
In other words, information we publish to the blackboard

C.1.4.11 CVirtualWorld* CAnimusCharacter::m_pVirtualWorld [protected]

a pointer to the virtual world
The documentation for this class was generated from the following file:

- AnimusCharacter.h

C.2 CAnimationThreadManager Class Reference

`#include <AnimationThreadManager.h>`

C.2.1 Detailed Description

This class handles the administration of animation threads.

It ticks all current threads, adds new ones, deletes those that must be deleted and performs animation of a character.

Usage:

1. Declare the object:

```
CAnimationThreadManager manager;
manager.create(this,exceptions,AMMSG_ENDSEGMENT|AMMSG_ENDSEQUENCE);
```

2. We can change the manager settings at any time

```
manager.setOptions(!AMMSG_ENDSEGMENT);
```

3. We can use the manager to begin a new animation:

```
manager.beginAnimation(messages);
manager.configureAnimation(from,fweight,to,tweight,efps,fps,secs,easing,options);
manager.saveAndLink();

or...

manager.setKeyframe(from,fweight,to,tweight,efps,fps,secs,easing,options);

or...

manager.setKeyframe(someAnimationThread);
```

4. At any time we can get access to the current animation

```
LPANIMATIONTHREAD currentAnim = getUnsavedAnimation();
```

5. We tick the animation with this function

```
manager.tick();
```

6. We can rewind the animation

```
manager.rewind();
```

7. We can clear the current animation

```
manager.clearAnimations();
```

8. we can clear all animations except the last one

```
manager.clearButLast();
```

Notes:

- The ability to save our current animation and temporarily set another animation can be changed later to the ability of having more than one animation concurrently running at the same time.
- When a virtual pose is created and an id is not provided, the assigned id is when we encounter a pose not anymore in use with this id, the manager will automatically delete it
- Process for recursion of animation sequences (gestures) goes like this:

1. Push the current sequence to save it
2. Set the manager running options (messages, looping, etc)
3. Set the new list, or construct a new list. If we did set a new list, call setInitialPose(...) to set the transition pose, and rewind the animations. If we did not set a new list we can use getSequence to retrieve the last created sequence and store it in the gesture object
4. After playing, manually pop the sequence if AM-CONF_LOOPGESTURE was specified. Otherwise the animation pops automatically.

About Bezier Interpolations:

The manager can take 2 threads with coincident internal edges and create a single segment with a softer interpolation. To do this:

1. Add two animations with any of the given options
2. Call bezierMerge(void) to merge the last two animations.

You must insert 2 more animations before calling bezierMerge again

The manager will discard the second thread since the first one will perform the complete animation. Times are added and easing is averaged.

The following options are available to configure an animation:

- **ATHREAD_BLENDED** This is a rigid blended animation thread
- **ATHREAD_SEQUENTIAL** A rigid sequential animation thread
- **ATHREAD_RIGID** A single rigid animation thread
- **ATHREAD_FLEXIBLE** A flexible animation thread
- **ATHREAD_LOOP** Loop this animation
- **ATHREAD_LOCKED** This animation cannot be interrupted

- **ATHREAD_SPLINE** This animation uses a third state and weight as control for bezier blending. Usually the way to set this is trough the bezierMerge function.
- **ATHREAD_VIRTUAL** Use as a parameter to redirect() when the new pose doesn't have nodes and it is desired to create a virtual pose between the current position and such pose. In this case the weight must be between 0-1

Animations can be set to trigger the following messages

- **ATHREADMSG_ONLOOP** We begin an animation loop
- **ATHREADMSG_ONEND** An animation has ended
- **ATHREADMSG_ONBREAK** Someone broke our animation

C.2.2 Member Function Documentation

C.2.2.1 LPANIMATIONTHREAD CAnimationThreadManager::beginAnimation (UINT *msg* = 0)

begins a new animation

C.2.2.2 void CAnimationThreadManager::bezierMerge (bool *useSnapshot* = false)

gets the 2 last inserted animations and creates a bezier blending

C.2.2.3 void CAnimationThreadManager::cleanLinkPose ()

Use when we need to store a pose in the m_pLinkPose member.
If the previous pose was virtual and volatile, we erase it here

C.2.2.4 void CAnimationThreadManager::clearAnimations (LPANIMATIONTHREAD *except* = 0)

clear the animation queue.
If we pass a thread as parameter the given thread is NOT erased (in case it exists)

C.2.2.5 void CAnimationThreadManager::clearButLast (void)

delete all except last one

C.2.2.6 void CAnimationThreadManager::configureAnimation (LPOSE *pFrom*, float *fWeight*, LPOSE *pTo*, float *tWeight*, UINT *efps*, UINT *fps*, float *secs*, float *easing*, UINT *options*)

configures the last created animation

C.2.2.7 void CAnimationThreadManager::create (CCharacterPoseEngine * *parent*, CExceptionManager * *emanager*, UINT *options*)

creates a new manager

C.2.2.8 LPANIMATIONTHREAD CAnimationThreadManager::getCurrentAnimationOrigin (void)

gets the the start pose belonging to the animation we're currently playing. There is no guarantee that an id exists though.

C.2.2.9 void CAnimationThreadManager::getLastPose (LPOSE & *lastPose*, float & *lastWeight*)

retrieves the last scheduled pose and its information

C.2.2.10 CDoubleTemplateList<CAnimationThread>* CAnimationThreadManager::getSequence (void)

returns the current animation sequence

C.2.2.11 void CAnimationThreadManager::onMessage () [inline]

Message handler.

C.2.2.12 void CAnimationThreadManager::popSequence (void)

restores the original animation sequence

C.2.2.13 void CAnimationThreadManager::pushSequence (void)

backs-up the current animation sequence

C.2.2.14 bool CAnimationThreadManager::recomendRedirection (void)

Returns true if redirection seems like a god idea. False if the minimum tick quota hasn't been met. Redirecting if this function returns false might not look very good

C.2.2.15 void CAnimationThreadManager::rewind (void)

sets the lerp value of all animations to 0

C.2.2.16 void CAnimationThreadManager::saveAndLink (void)

inserts the newly created animation into the list

C.2.2.17 void CAnimationThreadManager::setAnimationSequence (CDoubleTemplateList< CAnimationThread > * *pSequence*)

sets the current animation sequence.

One should call **pushSequence(void)**(p. 111) before using this function. This function also reasigns the last pose

C.2.2.18 void CAnimationThreadManager::setInitialPose (LPOSE *aPose*, float *aWeight*)

overrides the first pose so that animation runs smoot from one gesture to another. This function must be called AFTER setting a new animation sequence

C.2.2.19 void CAnimationThreadManager::setKeyframe (LPANIMATIONTHREAD *aThread*)

sets an externally created keyframe

C.2.2.20 LPANIMATIONTHREAD CAnimationThreadManager::setKeyframe (LPOSE *pFrom*, float *fWeight*, LPOSE *pTo*, float *tWeight*, UINT *efps*, UINT *fps*, float *secs*, float *easing*, UINT *options*)

if we want to do all the 3 mentioned functions at the same time, use this one. Pass null as the pose for using the last pose as link pose

C.2.2.21 void CAnimationThreadManager::setMinimumTicks (unsigned int *mTick*) [inline]

sets the minimum ticks an animation has to play before it can be interrupted.
if we put this too hight, we force the thread to complete.

C.2.2.22 void CAnimationThreadManager::setOptions (UINT *options* = 0)

sets the manager options

C.2.2.23 void CAnimationThreadManager::tick (void)

tick all animations

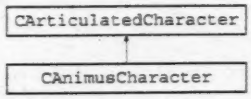
The documentation for this class was generated from the following file:

- AnimationThreadManager.h

C.3 CArticulatedCharacter Class Reference

`#include <ArticulatedCharacter.h>`

Inheritance diagram for CArticulatedCharacter::



C.3.1 Detailed Description

This is one of the major classes for the expression layer implementation.

It loads the required information from two sources: the mesh, bones and keyframes contained in the definition x file, and the organization and pose configuration contained in the pose xml file.

Every character should either contain or inherit from this class in order to be able to appear an perform animations

Usage:

1. To create and initialize:

CArticulatedCharacter cat; cat.create("cat.x", "catPoses.xml");

Life Cycle:

1. update pose if necesary on the pose engine
2. tick the pose engine when the application ticks us
3. render

C.3.2 Member Function Documentation

C.3.2.1 bool CArticulatedCharacter::create (char * *name*, char * *xFile*, char * *poseFile*, LPDIRECT3DDEVICE8 *pd3ddevice*)

initialization function

C.3.2.2 void CArticulatedCharacter::onMessage () [inline]

process messages

Reimplemented in **CAnimusCharacter** (p. 105).

C.3.2.3 CCharacterPoseEngine* CArticulatedCharacter::poseEngine (void) [inline]

access the pose engine

C.3.2.4 void CArticulatedCharacter::render (LPDIRECT3DDEVICE8 *pd3ddevice*, LPD3DXMATRIX *pworldmat*)

render the scene

C.3.2.5 void CArticulatedCharacter::tickArticulatedCharacter ()

tick the character structure

C.3.3 Field Documentation

C.3.3.1 char* CArticulatedCharacter::m_cName [protected]

name of this character

C.3.3.2 CExceptionManager CArticulatedCharacter::m_exceptions [protected]

an exception manager

C.3.3.3 CCharacterPoseEngine CArticulatedCharacter::m_poseEngine [protected]

the pose engine used to animate the character

C.3.3.4 CSkinMesh CArticulatedCharacter::m_skinMesh [protected]

the visible mesh

The documentation for this class was generated from the following file:

- ArticulatedCharacter.h

C.4 CCharacterPoseEngine Class Reference

`#include <CharacterPoseEngine.h>`

C.4.1 Detailed Description

This is the main class for the pose engine.

Contains all the structures required to blend poses, and walks among them to display the character as it should be displayed

The animation cycle should go like this:

1. Make all required transformations
2. tick with the current time
3. render

TO CREATE AN ANIMATION:

Use the animation creation techniques explained in AnimationThread.h accessing the animation manager with getAnimationManager(void)

C.4.2 Member Function Documentation

C.4.2.1 void CCharacterPoseEngine::addBoneChain (LPBONECHAIN *bChain*) [inline]

adds a bone chain

C.4.2.2 void CCharacterPoseEngine::addGesture (CGesture * *gesture*) [inline]

adds a gesture

C.4.2.3 void CCharacterPoseEngine::addPose (CPose * *pose*) [inline]

adds a pose

C.4.2.4 CAnimationThreadManager* CCharacterPoseEngine::animationManager (void) [inline]

returns the animation manager

C.4.2.5 bool CCharacterPoseEngine::bezierBlending (LPOSE *pFrom*, float *fFrom*, LPOSE *pTo*, float *fTo*, LPOSE *pControl*, float *fControl*, float *lerp*)

Bezier blending technique that blends from pose1 to pose2, taking poseN as a bezier control point.

ax,ay,bx,by,cx and cy must be calculated somewhere else.

C.4.2.6 bool CCharacterPoseEngine::bezierLerpBlending (LPOSE *pFrom*, float *fFrom*, LPOSE *pTo*, float *fTo*, LPOSE *pControl*, float *fControl*, float *t1*, float *t2*, float *lerp*, CDoubleTemplateList< CVirtualBoneChain > * *snapshot* = 0)

bezier-triangulated blending.

Instead of taking a bezier point, we do a triple interpolation for each bone. This way we preserve proportions and still get the correct lerp. If the animation is flexible, send the pose snapshot. Otherwise just ignore

C.4.2.7 bool CCharacterPoseEngine::blendBone (UINT *key1*, UINT *key2*, LPBONE *aBone*, float *lerp* = 0.5)

given a bone and 2 keys, sets the transformation matrix of the bone to the lerp value.

C.4.2.8 bool CCharacterPoseEngine::blendPose (LPOSE *pPose1*, float *w1*, LPOSE *pPose2*, float *w2*, float *lerp*)

given 2 poses with coinciding bones and a lerp value, this function sets a blend pose As a result the bones in pose 1 are affected by the transformations.

If either pose has no nodes the weight parameters are ignored

C.4.2.9 bool CCharacterPoseEngine::create (CArticulated-Character * *pParent*, CSkinMesh * *pSkinMesh*, CExceptionManager * *exceptions*, char * *poseFile*)

load the structure from an xml file

C.4.2.10 LPOSE CCharacterPoseEngine::createVirtualPose (LPOSE *pFrom*, float *wFrom*, LPOSE *pTo*, float *wTo*, float *lerp*, char * *name* = 0)

creates a virtual pose in realtime.

If a name for the pose is provided, the pose is added to the main list. Otherwise the pose will be deleted when the engine finishes using it. The lerp value indicates the interpolation place between pFrom and pTo

C.4.2.11 bool CCharacterPoseEngine::flexiblePoseBlending (CDoubleTemplateList< CVirtualBoneChain > * *snapshot*, LPOSE *aPose*, float *weight*, float *lerp*)

flexible blending takes a list of initial keys and a destination pose and weight.

C.4.2.12 CAnimationNode* CCharacterPoseEngine::getAnimationNodeByName (char * *anName*)

retrieve an animation node by its name

C.4.2.13 CBoneChain* CCharacterPoseEngine::getBoneChainByName (char * *cName*)

retrieve a bone chain by its name

C.4.2.14 CGesture* CCharacterPoseEngine::getGestureByName (char * *gname*)

retrieve a gesture by its name

C.4.2.15 CPose* CCharacterPoseEngine::getPoseByName (char * *pname*)

retrieve a pose by its name

C.4.2.16 void CCharacterPoseEngine::getSafeKey (LPSKELETON *theSkeleton*, LPBONE *theBone*, LPVIRTUALKEY & *theKey*)

always returns the NEXT key of a bone.

If the bone is virtual it returns its virtual part, otherwise it returns its corresponding defined part. The key must be a valid CVirtualKey object the skeletons must have been initialized with CSkeleton::prepareBones(void) this function walks the list of elements of the skeleton, so one has to AVOID accessing its elements with other functions like getNextBoneKey(...) and such.

C.4.2.17 void CCharacterPoseEngine::loadAndPlayGesture (char * *cPose*, UINT *efps*)

load and execute a simple blending gesture.

This will destroy any previous animation

C.4.2.18 void CCharacterPoseEngine::onMessage ()

Process incomming messages from various sources.

C.4.2.19 bool CCharacterPoseEngine::rigidWalkTo (LPOSE *aPose*, UINT *fps*, UINT *sec*, float *weight*, float *easing*, bool *blend*)

Interpolates into a given pose, starting from the current pose.

At the end the current pose becomes the destination pose

C.4.2.20 bool CCharacterPoseEngine::sequentialBone (UINT *key1*, UINT *key2*, LPBONE *aBone*, float *lerp*)

given a bone and 2 keys we look at their timesteps and set an adequate time according to the lerp value.

We then choose an intermediate key instead of performing global interpolated blending

C.4.2.21 bool CCharacterPoseEngine::sequentialPose (LPOSE *pPose1*, float *w1*, LPOSE *pPose2*, float *w2*, float *lerp*)

given 2 poses and the lerp value, we interpolate across a set of intermediate poses

C.4.2.22 void CCharacterPoseEngine::setRigidPose (CPose * *aPose*, float *weight* = 0)

immediately sets the indicated pose

C.4.2.23 void CCharacterPoseEngine::tick ()

ticks the engine. constant fps is read from the configuration file

C.4.2.24 bool CCharacterPoseEngine::virtualAndRigidBlend (LPBONE *aBone*, LPVIRTUALKEY *vKey*, UINT *aKey*, bool *bVirtualToRigid*, float *lerp*)

we blend from the virtual key of the bone to the specified key, OR from the real key to the virtual key, depending on the virtualToRigid param.

C.4.2.25 bool CCharacterPoseEngine::virtualToVirtualBlend (LPBONE *aBone*, LPVIRTUALKEY *vKey1*, LPVIRTUALKEY *vKey2*, float *lerp*)

virtual-to-virtual blending requires a bone and 2 virtual keys

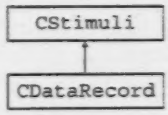
The documentation for this class was generated from the following file:

- CharacterPoseEngine.h

C.5 CDataRecord Class Reference

`#include <DataRecord.h>`

Inheritance diagram for CDataRecord::



C.5.1 Detailed Description

A DataRecord is the tuple of an object and its published stimuli chain at a given time.

This is what the sensor perceives and the fundamental input to the perception system of any entity.

An entity creates a data record by supplying a pointer to itself and the stimuli chain, or sucesively adding stimuli information

Usually one manipulates features and stimuli directly from this class, but sometimes it might be required to create the stimuli chain and then add it here.

C.5.2 Constructor & Destructor Documentation

C.5.2.1 CDataRecord::~CDataRecord (void)

destructor

C.5.3 Member Function Documentation

C.5.3.1 ENTITY CDataRecord::getEntity ()

Retrieve the entity.

C.5.3.2 CStimuli* CDataRecord::getStimuli ()

return ourselves as a stiimuli

C.5.3.3 CDataRecord& CDataRecord::operator= (CDataRecord & *dr*)

copy the information of another record

C.5.3.4 void CDataRecord::setEntity (CEntity * *entity*)

Assign an entity.

C.5.3.5 void CDataRecord::setStimuli (CStimuli & *s*)

Retrieve a particular stimuli.

C.5.3.6 void CDataRecord::threshold (CStimuli * *s*, float *theta* = -1) [static]

static thresholding
The documentation for this class was generated from the following file:

- DataRecord.h

C.6 CDriver Class Reference

`#include <Driver.h>`

C.6.1 Detailed Description

A driver is the place where certain intern feature is kept, accessed and modified by a particular process in the cognition layer.

Feature magnitudes are perceived in the perception layer and sometimes detonate cells with adequate thresholds. We could declare a bunch of CFeature objects in our agent and then play with them, but we define a CFeature-derived object instead because we have the advantages of a good data abstraction.

The magnitude of a driver will depend on:

1. The value of its feature 2. The value of the drivers with whom it relates 3. The sum of multiplications of weights across related drivers

This way, some drivers can inhibit or enhance others

Usage

Drivers can be used in a variety of ways, examples of configurations would include:

- Independent drivers: A driver keeping track of a magnitude, and a series of objects that monitor the messages of the driver
- Administered drivers: A driver whose magnitude changes in time, according to an algorithm present on a driver manager
- Driver network: A group of drivers linked together. Their magnitudes are influenced
- Hibrid: a combination of the above

Each driver contains a list of input arcs consisting on a pointer to other drivers and a weight. At time t, the value of this drive is equal to:

The drive magnitude m + [sum(magnitude of driver i x weight from driver i)]

Example of setting two drivers, heat affecting pain:

1. Define the drivers

2. Make pain a function of heat, pain = |pain| + |heat|*1.3 pain.setArcFrom(&heat,1.3);

3.a Set a cycle that stops if pain is too much

```
do
{
   applyFlame();        // set some stimuli over the pain driver
}while(!pain.getMessages()&DM_THRESHOLD );
```

3.b optionally, set a callback object that will receive interesting messages:

registerParent(someObject);

Then the onMessages function of the parent will be called:

```
onMessages()
{
   ....
   case DM_THRESHOLD:
      stopFlame();
      break;
   ...
}
```

One can register as many objects as desired with a single driver. This way anyone interested will receive notifications from the driver itself without having to have a reference.

Please note that the CDriver class provides means for setting up the 'phisical' aspect of a driver system network, but a latter class probably named CDriver-Network will be in charge of managing driver interaction, collecting messages, and most important: calculating influence magnitudes (walking the driver network)

Modifying a driver

1. Set the value of a driver with any of the overloaded modificators, this resets the updating flag of the driver and prepares information to be updated 2. call getValue or getFlags

ONE MUST NEVER ACCESS THE VALUE DIRECTLY!

The driver will recurse any influencing driver automatically

Flag triggering

In order to effectively trigger flags, the driver must be modificated with the overloaded +=, *= and /= operators, and the value must be retrieved with the getMagnitude() function. Otherwise it is not guaranteed that the driver or its influence drivers act on the magnitude.

Remember: If the CDriver object is a pointer, call these operators like this:

```
CDriver  a;
CDriver *b;

a+=3;   // correct. a is NOT a pointer
*b+=5;  // correct. b IS a pointer
b+=5;   // incorrect. b is now GARBAGE
```

Decay and gain

Some of the authors that use drivers define a decay value or function in the drivers so that magnitude tends to modify by itself over time. Here drivers have a slightly different approach. There must be another process specialized in controlling certain kind of feature, and that is the process that, linked to the driver, will diminish or increase its magnitude. This allows for more complex conditional manipulation, example: with decay function, anger will

slowly diminish. With anger driver manager, anger will completely go away if character is given a present or falls asleep. See the diagram:

a) conventional driver:

CDriver(anger) decay over time ——> sensors

b) my approach:

CDriverManager(anger)appraises events and controls—>CDriver(anger) —> sensors

A driver manager will have access to the driver and can modify it at any given time trough the driver network layer, so that the cycle at time t is:

[driverNetwork: updates]->[managers: read and modify drivers] -> repeat [other processes: read and modify]

Some drivers might not need a manager at all, since they just are the result of the interaction of other drivers and ocassional input (like pain, for example)

C.6.2 Constructor & Destructor Documentation

C.6.2.1 CDriver::CDriver (void)

default constructor

C.6.2.2 CDriver::CDriver (CDriver & *d*)

copy constructor

C.6.2.3 CDriver::CDriver (char * *uid*, FEATURE *feature*, float *startValue*, float *stasis*, float *threshold*, float *umbral*, unsigned int *events*)

receives the feature, current value, statis value, threshold value and list of events we want the driver to respond

C.6.2.4 CDriver::~CDriver (void)

destructor

C.6.3 Member Function Documentation

C.6.3.1 float CDriver::getMagnitude ()

to retrieve magnitude...

C.6.3.2 unsigned int CDriver::getMessages ()

to retrieve messages

C.6.3.3 void CDriver::operator+= (float *f*)

to modify the value of a drive we use any of the following test operators.
Driver modification will not apply until we tell the driver to update. All modification functions will reset the m_changed flag

C.6.3.4 void CDriver::registerParent (MESSAGERECEIVER *parent*)

Register a CMessageReceiver-derived object that will handle the things that happen to this driver.

C.6.3.5 void CDriver::reset (void)

reset the driver by setting to stasis value

C.6.3.6 void CDriver::setArcFrom (CDriver * *from*, float *weight*)

Sets an arc from the specified driver to this one.

C.6.3.7 void CDriver::update ()

do recursion in order to get the most up-to-date magnitudes and messages.

C.6.3.8 void CDriver::validateChange (float *old*)

check what the change is actually going to do.
This is to be performed before modifying the magnitude

C.6.3.9 void CDriver::validateThreshold ()

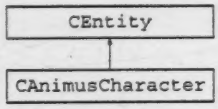
check threshold and stasis rules.
This can be done after setting the magnitude
The documentation for this class was generated from the following file:

- Driver.h

C.7 CEntity Class Reference

`#include <Entity.h>`

Inheritance diagram for CEntity::



C.7.1 Detailed Description

CEntity,.

This is the base class for the perception layer of everything that exists inside the virtual world, be it an unanimated object, a virtual character, an avatar or anything else. From this class, particular 'entities' will derive and add specific functionallity.

Some characteristics of the entities:

- Each entity has a unique id
- Each entity can be queried for a particular feature, but the result might be a true, null or noise-affected response
- Query outputs are stimuli objects instead of data records because it is another entity who, using the m_pEntity field of some other data record, directly makes the query.
- When making a query, a null result means the object doesn't have anything to say about the property (i.e., it cannot be known), whereas a cero magnitude actually represents the published magnitude of such feature.

Implementation details:

- Any CEntity-derived class shall implement the virtual functions to its own functionallity
- In addition, any CEntity-derived classes will implement the 3 main layers of a broad agent (perception|cognition|expression), so the final entity-type class is a fully functional virtual character, something like this:

CEntity->CAnimalEntity->CDog

CAnimalEntity implements perception, cognition and expression for an animal-like thing, while CDog provides specific particularizations for those layers to create a character that behaves like a dog

Differences between entity messaging and internal-organs messaging:

- Entities send messages by means of **CDataRecord** structures, while internal organs like drivers send information using the CMessage structure.

C.7.2 Member Function Documentation

C.7.2.1 char* CEntity::getId () [inline]

return our id

C.7.2.2 ETYPE CEntity::getType () [inline]

return our type

C.7.2.3 virtual void* CEntity::publish () [pure virtual]

at a given time t, an entity must be able to publish something.
This function is likely to be called by the virtual world administrator and not any other entity

C.7.2.4 virtual CFeature* CEntity::query (FEATURE *f*) [pure virtual]

query a single feature about this entity. Result can be null

C.7.2.5 virtual CStimuli* CEntity::query (CStimuli & *s*) [pure virtual]

query a series of features about this entity. Result can be null

C.7.2.6 ETYPE CEntity::strToType (char * *str*) [static]

get the entity type
The documentation for this class was generated from the following file:

- Entity.h

C.8 CMemoryUnit Class Reference

`#include <MemoryUnit.h>`

C.8.1 Detailed Description

CMemoryUnit is the manager of all the other memory-related classes:.

- CFrame
- CTimeline
- CKeyframe and CLeadKeyframe
- CMemoryLayer

It is a wrapper and administrator of everything that happens in those objects. It also respons queries, adds elements and maintains information

Usage:

1. Construct the CMemoryUnit object
2. To add a layer: specify the character for such layer. One can keep the returned pointer as a reference to the layer
3. To add a keyframe specify the layer, the stimuli and the time (specify no time if you're adding at the end)

```
// adds at t = 10
mUnit.addKeyframe(someLayer,someStimuli,10);
// adds at last frame
mUnit.addKeyframe(someLayer,someStimuli);
// help getting the layer
mUnit.addKeyframe(getLayer(someCharacter), someStimuli);
```

4. To retrieve a complete frame use getFrame(time). let time = 0 to retrieve the current frame. Use **getNextFrame()** to retrieve the frame and advance the current frame

```
frame10 = mUnit.getFrame(10);
lastFrame = mUnit.getFrame();

mUnit.timelineHead();
while(mUnit.getNextFrame(thisFrame))
{
  thisFrame BLA BLA BLA
}
```

5. To retrieve a complete layer, specify the character

```
myCharacterLayer = getLayer(myCharacter);
```

6. To retrieve a particular keyframe data, specify layer and time. Leave time at 0 for the last keyframe of the layer

```
myStimuli = mUnit.getKeyframeInfo(someLayer,someTime);
myStimuli = mUnit.getKeyframeInfo(getLayer(myCharacter),someTime);
```

7. To retrieve the character that owns a keyframe, use **getCharacter(CDoubleTemplateElement<CKeyframe> *key)**

```
someCharacter = getCharacter( frame10.getNextElement() );
```

8. Use the special edition function forceGetKeyframe(layer,t) to retrieve the keyframe of the layer. If no keyframe exists, it geats created (a diferencia de getKeyframeInfo)

Prunning and optimization

- If the function shortTerm(set,gap,del) is called with set=true, every frames the class will delete frames without including the first, in order to keep the memory optimized and avoid conglomeration. If gap = 1, the memory unit will keep, at all times, only the first and last frames. must be equal or less than .

shortTerm MUST NOT be set if the memory will not be sequential (i.e., we will insert keyframes at non-sequential times)

C.8.2 Member Function Documentation

C.8.2.1 void CMemoryUnit::addKeyframe (CMemoryLayer * *layer*, CStimuli & *s*, unsigned int *t* = 0)

adds a keyframe.

WARNING: Nothing prevents you from adding multiple keyframes for the same layer at the same time

C.8.2.2 CMemoryLayer* CMemoryUnit::addLayer (CCharacter * *character*)

adds a layer for a particular character

C.8.2.3 void CMemoryUnit::checkPrune (void)

checks and prunes the timeline

C.8.2.4 void CMemoryUnit::clearMemory ()

clear everything, WARNING: we DELETE all elements

C.8.2.5 void CMemoryUnit::deleteLayer (CMemoryLayer * *l*)

Use this function to delete a particular layer.

C.8.2.6 CStimuli* CMemoryUnit::forceGetKeyframe (CMemoryLayer * *layer*, unsigned int *t*)

get the stimuli of keyframe in the given layer and time.

If no keyframe is found at that time, a new one is created and its empty CStimuli* object returned.

C.8.2.7 CCharacter* CMemoryUnit::getCharacter (CDoubleTemplateElement< CKeyframe > * *key*)

get the character of the proposed keyframe

C.8.2.8 CStimuli* CMemoryUnit::getClosestKeyframe (CMemoryLayer * *layer*, unsigned int *t*)

gets the corresponding stimuli at time t.

If no stimuli exists at that time, the function returns the closest one

C.8.2.9 CFrame* CMemoryUnit::getFrame (unsigned int *t* = 0)

retrieves some frame, or null if such frame doesn't exist.

cero to get last frame

C.8.2.10 CStimuli* CMemoryUnit::getKeyframeInfo (CMemoryLayer * *layer*, unsigned int *t* = 0)

get some stimuli, unless any of the params is wrong (then we receive null).

If t=0 we retrieve the last keyframe of the layer

C.8.2.11 CMemoryLayer* CMemoryUnit::getLayer (CEntity * *entity*)

this version only requires an entity to return the corresponding layer, or null if the entity has no layer at all

C.8.2.12 CMemoryLayer* CMemoryUnit::getLayer (CCharacter * *character* = NULL)

get some layer, or null if layer is not found.

pass null to receive the current layer and advance the head to the next one. Use in this way to walk trough all layers in the following way:

```
memoryObject.m_lLayers.prepare();
while( someLayer = memoryObject.getLayer() ){  ////do things//// };
```

get the next frame, or false if last frame has been returned already

C.8.2.13 void CMemoryUnit::getTimelineRange (unsigned int & *start*, unsigned int & *end*)

get the timeline range

C.8.2.14 bool CMemoryUnit::loadFromXML (char * *filename*)

load from an xml file

C.8.2.15 void CMemoryUnit::saveAsXML (char * *filename*)

dump to xml file

C.8.2.16 void CMemoryUnit::shortTerm (bool *set* = true, int *gap* = 10, int *del* = 10)

sets automatic prunning, transforming this memory module into a short-term memory

C.8.2.17 void CMemoryUnit::timelineHead ()

move the pointer to the beginning of the timeline

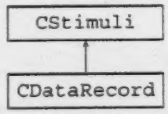
The documentation for this class was generated from the following file:

- MemoryUnit.h

C.9 CStimuli Class Reference

`#include <Stimuli.h>`

Inheritance diagram for CStimuli::



C.9.1 Detailed Description

A stimuli chain is the advertised information of some entity at a given time.

It consists on a series of features (one if this is a driver) and its magnitudes.

The stimuli chain, along with a pointer to the entity that produced it constitutes a DataRecord; which is to be processed by the perception layer.

this is a double linked list to allow simple elimination of elements

C.9.2 Member Function Documentation

C.9.2.1 void CStimuli::addFeature (CFeature & *f*)

adds a single feature to this stimuli

C.9.2.2 void CStimuli::addFeature (CStimuli & *s*)

adds a stimuli chain to this stimuli

C.9.2.3 void CStimuli::addFeature (FEATURE *f*, float *m*)

adds a feature to the end of the list.

For convenience, this creates the feature object

C.9.2.4 bool CStimuli::contains (CFeature & *f*)

Checks if some feature is present.

f->m_feature is tested and if a match is found, the result is returned in f->m_data, otherwise the return falue is false. This function is not for changing the value, but for querying the stimuli chain. If it is desired to test for the existence of a stimulus one should use getFeature(feature) instead

C.9.2.5 void CStimuli::copyStimuliChain (CStimuli & *s*)

equal two stimuli chains.

This deletes the current stimuli chain even if the one proposed is empty

C.9.2.6 double CStimuli::cosineCorrelation (CStimuli * *c*, CStimuli * *t*) [static]

return the cosine similarity between two data records.

T is compared against C usually c is the contained stimuli, and t is the new stimuli

C.9.2.7 void CStimuli::deleteAll ()

delete all units

C.9.2.8 float CStimuli::getAverage ()

the average value

C.9.2.9 CFeature* CStimuli::getFeature (FEATURE *f*)

the feature is not found. If the feature is repeated, returns the first occurrence.

C.9.2.10 float CStimuli::getMagnitude ()

get the euclidian magnitude of the stimuli vector, that is m = sum(i=0->m_lFeatures.m_iSize)feature i

C.9.2.11 bool CStimuli::getNextStimuli (CFeature *& *f*)

this version returns a pointer to the feature

C.9.2.12 bool CStimuli::getNextStimuli (CFeature & *f*)

recurse elements for retrieval.

Structure goes as follows: CFeature myFeature; sChain.prepare(); while(sChain.getNextStimuli(myFeature)){ ... };

C.9.2.13 CStimuli* CStimuli::intersect (CStimuli & *s*)

returns a sublist with shared features, or null if no feature is found.

Note that the magnitudes of the returned features are those found on this list and not on s

C.9.2.14 void CStimuli::or (CStimuli & *s*)

Or's the binary information in s with our own information, adding features of s not contained in our stimuli and setting their value to 1 if any of them contain a value at all.

C.9.2.15 void CStimuli::prepare ()

prepare for recursion

C.9.2.16 void CStimuli::removeLastReturned ()

removes the last returned stimuli

C.9.2.17 void CStimuli::removeStimuli (CFeature * *toRemove*)

removes a node with equal feature

C.9.2.18 void CStimuli::setCurrent (OPERATION *p*, float *factor*)

Recurse elements for modification.

Begin with prepare, then cycle using setCurrent(OPERATION,factor) while atEnd is not true:

C.9.2.19 bool CStimuli::setFeature (FEATURE *f*, float *m*)

sets the specified feature to the specified value IF the feature is indeed contained in our chain.

Otherwise it returns false

The documentation for this class was generated from the following file:

- Stimuli.h

C.10 CVirtualWorld Class Reference

`#include <VirtualWorld.h>`

C.10.1 Detailed Description

Manages the blackboard used to communicate betweeen characters, has the list of agents present in the environment and queries them for information.

For more information, see conventions.txt

C.10.2 Member Function Documentation

C.10.2.1 void CVirtualWorld::create (float *ufrec*, CExceptionManager * *pExceptionManager* = NULL)

creation function.

Receives the update frequence and the exception manager. If we don't pass a manager, the class creates its own.

C.10.2.2 CStimuli* CVirtualWorld::getAgentStimuli (CAnimusCharacter * *pAgent*)

returns the stimuli in the blackboard for a certain agent.

if the agent doesn't have a record it returns null

C.10.2.3 LPBLACKBOARDITERATOR CVirtualWorld::getBlackboard (void)

returns an iterator for the blackboard

C.10.2.4 void CVirtualWorld::onRender (LPDIRECT3DDEVICE8 *pd3dDevice*, LPD3DXMATRIX *pworldmat*)

Every time the application is ready to render.

the pworldmat matrix is normally the identity matrix unless we want to offsett all of the characters

C.10.2.5 void CVirtualWorld::onUpdate (float *eTime*)

Every time we want the world to update.

C.10.2.6 void CVirtualWorld::registerAgent (class CAnimusCharacter * *pAgent*)

registers a new agent

C.10.2.7 void CVirtualWorld::updateBlackboard () [protected]

function to update the blackboard

C.10.3 Field Documentation

C.10.3.1 CDoubleTemplateList<CDataRecord> CVirtualWorld::m_blackboard [protected]

A blackboard for keeping messages.

C.10.3.2 float CVirtualWorld::m_fWorldAccTime [protected]

accumulated time for world update

C.10.3.3 float CVirtualWorld::m_fWorldUpdateFrec [protected]

delta in time for world update

C.10.3.4 CDoubleTemplateList<class CAnimusCharacter> CVirtualWorld::m_lAgents [protected]

A list of all the characters currently present in the world.

C.10.3.5 CExceptionManager* CVirtualWorld::m_pExceptionManager [protected]

our exception manager

The documentation for this class was generated from the following file:

- VirtualWorld.h